AD-A235 911

HATFIELD POLYTECHNIC

NUMERICAL
OPTIMISATION
CENTRE

Optimisation Algorithms for Highly Parallel Computer Architectures.

L. C. W. Dixon
R. C. Price

December 1990

AD 1990

OPTIMISATION ALGORITHMS FOR HIGHLY PARALLEL COMPUTER ARCHITECTURES

DRAFT FINAL REPORT

BY

L. C. W. DIXON

&

R. C. PRICE

DECEMBER 1990

UNITED STATES ARMY

EUROPEAN RESEARCH OFFICE OF THE ARMY

LONDON, ENGLAND

CONTRACT NUMBER DAJA45-87-C-0038

HATFIELD POLYTECHNIC

Optimisation Algorithms for Highly Parallel

Computer Architectures

by

L. C. W. Dixon and R. C. Price

December 1990

Abstract

In this paper we consider the design of optimization algorithms to run efficiently on highly parallel computer architectures. Most efficient optimization algorithms require the calculation of first and possibly second derivatives of the objective function and where present the constraints. This task normally dominates all other tasks undertaken in solving a large sized problem. The other main task is usually the solution of a set of linear equations.

In this paper we describe our experience of solving these two tasks on parallel computer architecture. A sparse forward implementation of doublet and triplet automatic differentiation is described that enables both the gradient and hessian of objective functions to be accurately and cheaply solved. It is shown that when the function is partially separable this can be performed very effectively on a parallel machine.

The effect of solving sets of linear equations on a parallel system is also described, and the two then combined in effective algorithms for both unconstrained and constrained optimization.

Keywords

Automatic differentiation, unconstrained optimisation, nonlinear programming, parallel optimization, truncated Newton, preconditioned conjugate gradients.

Table of Contents

List of Figures

List of Tables

1. Introduction

For many years the scale of optimization problems that can be solved has been limited by the sequential nature of available computers. The availability of highly parallel architectures will significantly increase the size of problems that can be solved. Indeed this is already true of the limited parallel architecture systems now available. The solution of such problems is of course necessary for the SDI programme.

However, because all existing optimization software designed before 1983 was essentially designed for sequential computers, the design of parallel algorithms must be radically different and so the initial search on algorithm design for parallel computers can be undertaken on relatively small parallel systems.

Simultaneously with the move towards highly parallel architectures, the language ADA has become available including the features of user defined data types, operator overlay and parallel (concurrent) tasking.

The project aimed to utilise these features of ADA to design efficient optimization codes for use on highly parallel architectures.

Specifically the following four areas have been considered.

(1) To investigate the effect of the availability of user defined data types and operator overlay on the design of optimization software, bearing in mind the ultimate goal of use on a highly parallel computer architecture.

(2) To investigate the relative efficiency of ordinary direct and indirect iterative methods for solving $Ax = b$, $x \in R^n$ on a parallel computer system with P processors for various ratios of P:n. And to compare the efficiency of these methods with those based on interval arithmetic.

(3) To investigate the design of algorithms for solving the unconstrained optimization problem

$$\text{Min } F(x); \; x \in R^n \tag{1.1}$$

(4) To investigate the design of algorithms for solving the constrained optimization problem

$$\text{Min } F(x)$$

$$\text{s.t.} \quad e_i(x) = 0 \quad i = 1, \ldots, E \quad x \in R^n \quad E < n$$

$$h_j(x) \geq 0 \quad j = 1, \ldots, H \tag{1.2}$$

The project commenced by defining the data types vector and matrix, and then redefining the values of the operators +, -, *, so that the computer automatically performed the normal operations of linear algebra.

This implied that if the operation

$$y = A*x \tag{1.3}$$

was requested in the code, where A was of type matrix and x of type vector, then the computer produced the correct vector y with no further commands.

It was soon realised that this enabled the computer code for the conjugate gradient algorithm to be virtually identical to the mathematical statement as all the "do" loops that so clutter a FORTRAN program rapidly disappeared.

It was immediately apparent that it was just as possible to define data types, "sparse matrix" and "sparse vector", and define associate operators so that the operation

$$y = A*x$$

still gave the correct vector and made full use of the sparsity of the matrix when doing the arithmetic.

Codes written in this way are far easier to understand and check than the equivalent codes written in FORTRAN.

In a similar way a data type "interval" can be defined and its related operators implemented so that rounded interval algebra is performed to any precision. The realisation of the potential of such computer algebras led to the award of further funding from the NAB research initiative.

The current state of the software developed under that initiative is described in Bartholomew-Biggs (1990). which includes the calculation of accurate dot products and of matrix vector products accurate to the last significant place in the mantissa.

Once the concept of defining new algebras in ADA became accepted we realised that it was possible to define the algebra of doublets and triplets to calculate the gradients and hessians of objective functions and constraints. Our implementation is outlined in Section 2 where it is demonstrated that especially for partially separable functions this gives an accurate and fast method of calculating all such gradient vectors. We can regularly obtain the Jacobian of a 5000 x 5000 set of O.D.E.s in less than 50 times the cost of a function evaluation.

In our first interim report, Dixon (1987), a review was given of parallel methods for solving sets of linear equations and their application within optimisation algorithms.

In that report two different approaches to the design of parallel optimisation software were identified.

<u>Approach A</u>

The calculation of each objective function value $F(x)$ is divided into P parallel tasks. An approach which leaves the responsibility for the efficient use of parallelism in the hands of the user, and,

<u>Approach B</u>

An algorithm is designed so that it can accept P values of $F(x)$ or $\nabla F(x)$ simultaneously. An approach which leaves the responsibility for the efficient use of parallelism in the hands of the user.

Algorithms for both approaches are described in Section 3.

Our experience using interval methods to solve sets of linear equations was very disappointing (Parkhurst (1987)) and after performing the comparison between direct and indirect methods given in Tables 28 and 29 of Section 5 (Dixon & Maany (1987)) we tend to use preconditioned conjugate gradient algorithms for symmetric matrices and after considerable additional testing (Parkhurst (1990)), the preconditioned CGS method (Sonneveld(1986)) for nonsymmetric matrices.

In our third interim report (Dixon & Maany (1988)) our implementation of a truncated Newton code using Automatic differentiation, and matrix and vector packages in ADA was compared with a FORTRANN implementation of the same nominal algorithm. This demonstrated that our ADA implementation was only three times more expensive than the FORTRANN, even on problems of 3000 dimensions.

In the fourth interim report, Dixon, Maany and Mohseninia (1989a), further results were given for the sequential ADA code and the first results for the parallel implementation on the Sequent Balance. This is also discussed in Section 3.

The fifth interim report contained three papers presented at the IFAC Symposium on "Dynamic Modelling and Control of National Economies". In the first of these Dixon, Maany and Mohseninia (1989B) further developed the algebras of automatic differentiation and their inclusion in an unconstrained optimization algorithm. In the second Bartholomew-Biggs (1989) gave the first presentation of the use of automatic differentiation in a constrained optimization algorithm, whilst in the third Mohseninia (1989) described the use of parallel automatic differentiation to solve Navier Stokes equations by a finite element method on the Sequent Balance Multi processor machine. These results were described in more detail in the sixth interim report Dixon & Mohseninia (1990) in which a closer examination of the results indicated that whilst the parallel automatic differentiation was performing effectively the parallel linear equation solver was suffering due to the dominance of communication costs.

All our experience indicates that the ratio of the time required to

pass a floating point number between processors to the time required to multiply two floating point numbers is a critical parameter in the design of parallel software. The codes described in Section 3 on both the Sequent Balance and the transputer net would perform much better if the communication speed had been ten times faster.

Section 2 of this report is therefore devoted to a decription of automatic differentiation and Section 3 to our experience using unconstrained optimization algorithms on parallel processing systems. Section 4 is devoted to constrained optimization, while section 5 contains some speculations on future developments.

Some conclusions are given in Section 6.

2. The calculation of derivatives

2.1 Gradient and Hessian of a function f(x)

If we return first to the unconstrained optimisation problem (1.1); we will assume that the objective function f(x) is computable and that it can be expanded as a series of elementary operations of one or two variables

$$F_i(x_j, x_k).$$

So the objective function is given by

For $\quad i = n+1, \ldots, n+M$

$$x_i = F_i(x_j, x_k) \qquad j,k < i \qquad (2.1)$$

next i.

$$f = x_{M+n}$$

The elementary operations consist of addition, subtraction and multiplication of two variables supplemented by functions of one

variable i.e. sine, cosine, log, exponential, power, reciprocal etc.

Any computable objective function can be expanded into such a form if Boolian switches are ignored. For the purpose of this paper such switches will be ignored.

If we consider the well known Rosenbrock function

$$f(x) = 100(x_1^2 - x_2)^2 + (1-x_1)^2 \tag{2.2}$$

This could be expanded as

$$\begin{aligned} x_3 &= x_1^2 \\ x_4 &= x_3 - x_2 \\ x_5 &= x_4^2 \\ x_6 &= 100x_5 \\ x_7 &= 1-x_1 \\ x_8 &= x_7^2 \\ f = x_9 &= x_6 + x_8 \end{aligned} \tag{2.3}$$

So we see that this simple two dimensional problem contains seven operations. It is not unusual that for many practical problems $M>n^2$ or even n^3. This example also indicates that the order of the operations is often flexible and that the calculation may be indicated by a graph, which connects each operation with its data and emphasises any parallel computation possibilities in its calculation.

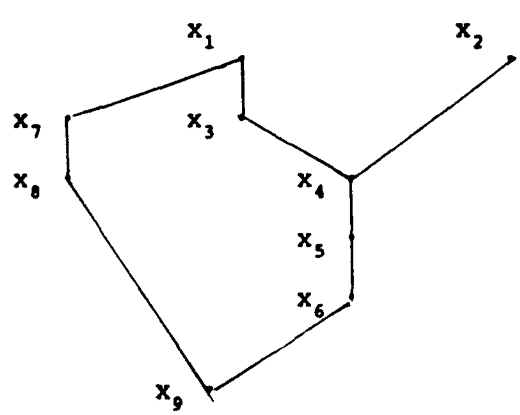


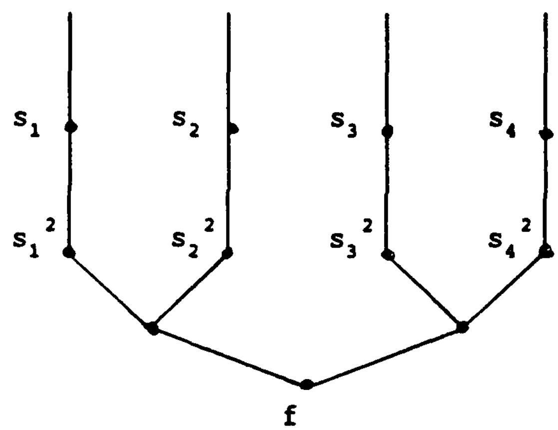
Figure 1. Graph of Rosenbrocks function

This graph emphasizes that the operations labelled 7 and 8 could have been undertaken at any stage in the calculation without altering the result. It also indicates that at most two processors could be used and that at least five sequential steps are required.

If we now consider any least squares problem

$$f(x) = \sum_{k=1}^{K} s_k^{\,2}(x) \tag{2.4}$$

then each subfunction $s_k(x)$ could have a graph like Rosenbrock's function and the subfunctions could be performed as independent calculations and the graph only linked at the end.



<u>Figure 2</u>

Any function with a number of parallel subtrees will be termed partially separable following Griewank and Toint (1981). Frequently but not necessarily each of the subfunctions s_k will depend upon a relatively small number of the independent variables x_i $i=1, \ldots, n$.

Now suppose we wish to find the gradient vector ∇f. How should we proceed? There are at least four alternative approaches.

M1 Analytic Differentiation; The formula for the gradient vector are derived analytically and entered into the computer. This was the method usually adopted by practitioners until quite recently. For Rosenbrock's function this is fairly simple as n=2 and the gradient vector is given by

$$\nabla f = \begin{pmatrix} 400\, x_1(x_1^2 - x_2) - 2(1-x_1) \\ \\ -200(x_1^2 - x_2) \end{pmatrix} \tag{2.5}$$

but as n and M increase this becomes increasingly tedious and mistakes are liable to occur.

M2 Numerical Approximation; One sided or central difference formula may be used to estimate the derivatives i.e. the approximation

$$\nabla f_i = (f(x+h\hat{a}_i) - f(x))/h \tag{2.6}$$

could be adopted.

This is of course simple to program and is often successfully used, it is however expensive as calculating ∇f now requires at least nM operations and is also dependent on choosing a suitable value of h.

M3 Symbolic Differentiation; Many mathematical aids such as MACSYMA will accept a FORTRAN listing of an objective function and produce a FORTRAN listing of the gradient. The difficulty of this approach is that such symbolic codes often produce inefficient FORTRAN and to date are often very restrictive on the complexity (M) of the problem that can be tackled. Grievank(1988) gives the listing of the gradient of the Helmholtz energy function given by MACSYMA. Such a listing then has to be evaluated at each iterative point $x^{(R)}$, and when the formula is lengthy it is not clear that this has any advantages over automatic differentiation.

M4 Automatic Differentiation; A full description of automatic differentiation is given in Rall(1981). Automatic differentiation is essentially the computer implementation of a new algebra, which is implemented by defining a new data type and overwriting the meaning of operators for that data type. The simplest such data type, the doublet, consists of the n+1 numbers $U=(u,\nabla u)$. Then given one or two doublets the meaning of the elementary operations is overwritten so, we obtain:-

$$\left.\begin{array}{lll} W = U+V & \rightarrow & (u+v,\ \nabla u+\nabla v) \\ W = U-V & \rightarrow & (u-v,\ \nabla u-\nabla v) \\ W = U*V & \rightarrow & (u\cdot v,\ u\nabla v+v\nabla u) \\ W = U^2 & \rightarrow & (u^2,\ 2u\nabla u) \\ W = U^{-1} & \rightarrow & (u^{-1},\ -u^{-2}\nabla u) \\ W = \log U & \rightarrow & (\log u,\ u^{-1}\nabla u). \end{array}\right\} \qquad (2.7)$$

As each of the operations in 2.1 is of this form we can obtain the derivatives by forward automatic differentiation in terms of doublets:-

For $i = n+1, \ldots, M+n$

$$(x_i,\nabla x_i) = F_i((x_j,\nabla x_j),(x_k,\nabla x_k)) \qquad j,k < i$$

next i

$$(f,\nabla f) = (x_{M+n},\nabla x_{m+n})$$

This approach is accurate but is bounded above by (2n+1)M operations due to the fact that a full vector ∇w is created at each step even though ∇w is usually sparse.

M4.1 Sparse Doublet and Sparse Triplet Arithmetic; An efficient implementation in ADA is described in Dixon, Maany, Mohseninia(1989) where the vector ∇w is stored in sparse form i.e. (Number of nonzero elements, position, value) so $\nabla x_i = (1,1,1)$ even if n=1000.

This method can be readily extended to the calculation of the second

derivatives by introducing the algebra of sparse triplets in which $T = (u, \nabla u, \nabla^2 u)$ and both ∇u and $\nabla^2 u$ are stored in sparse form. We will illustrate this by showing the calculation for Rosenbrocks function and then the Helmholtz energy function. As n=2 for Rosenbrock's function the illustration will ignore sparsity.

x_1	$(x_1;\ 1,0;\ 0,0,0)$
x_2	$(x_2;\ 0,1;\ 0,0,0)$
$x_3 = x_1^{\ 2}$	$(x_1^{\ 2};\ 2x_1,0;\ 2,0,0)$
$x_4 = x_3 - x_2$	$(x_4;\ 2x_1,-1;\ 2,0,0)$
$x_5 = x_4^{\ 2}$	$(x_5;\ 4x_1x_4,\ -2x_4;\ 4x_4 + 8x_1^{\ 2},2,-4x_1)$
$x_6 = 100x_5$	(all multiplied by 100)
$x_7 = 1 - x_1$	$(x_7;\ -1,0;\ 0,0,0)$
$x_8 = x_7^{\ 2}$	$(x_8;\ -2x_7,0;\ +2,0,0)$
$x_9 = x_6 + x_8$	$(x_9;\ 400x_1x_4-2x_7,-200x_4;$ $400x_4+800x_1^{\ 2},+2,200,-400x_1)$

which is the correct solution.

It will be noted that even on this very simple problem 19 of the 54 entries in the 9 triplets calculated are zero.

The Helmholtz energy function introduced by Grievank(1988) as a standard example for automatic differentiation is given by:-

$$f = RT \sum_{i=1}^{n} x_i \log \frac{x_i}{1-b^T x} - \frac{x^T A x}{\sqrt{8} b^T x} \log \frac{1+(1+\sqrt{2})b^T x}{1+(1-\sqrt{2})b^T x}\,. \qquad (2.8)$$

This can be divided into certain subfunctions, and in Table 1 we list the number of operations involved in the calculation of f, ∇f and $\nabla^2 f$ in sparse doublet and sparse triplet arithmetic.

	Function	Operations in F	∇F	$\nabla^2 F$
	$b^T x$	2n	n	0
	$1-b^T x$	1	0	0
	$1+(1+\sqrt{2})b^T x$	4	n	0
	$1+(1-\sqrt{2})b^T x$	4	n	0
	$1/(1-b^T x)$	1	n+2	n^2+3
	$1/1+(1-\sqrt{2})b^T x$	1	n+2	n^2+3
n times	$x_i/(1-b^T x)$	1	3n	$6n^2$
	$\log x_i/(1-b^T x)$	log	2n	$3n^2$
	$x_i \log x_i/(1-b^T x)$	1	3n	$6n^2$
	So the sum becomes	(2+log)n	(8n)n	$(15n^2)n$
	$x^T Ax$	$3n^2$	$4n^2$	n^2
	$1/b^T x$	1	n+2	n^2+3
	$x^T Ax/b^T x$	1	3n	$6n^2$
	log( )	log	2n	$3n^2$
	$\dfrac{x^T Ax/b^T x}{\log (\ )}$	1	3n	$6n^2$
	other	3	3n	$3n^2$
Total (approx)		(n+1)log+ $3n^2$	$12n^2$+ 17n	$15n^3$+ $22n^2$

Table 1.1

From the Table 1.1 we can see that very many terms contribute $O(n^2)$ operations to $\nabla^2 F$ but that the term $x^T Ax$ dominates F, whilst the term

$$\sum_{i=1}^{n} x_i \log x_i/(1-b^T x)$$

dominates $\nabla^2 F$ and appears to make it an $O(n^3)$ operation. As Maany observed when performing our tests in ADA, this term can be rearranged as

$$\sum_{i=1}^{n} x_i \log x_i/(1-b^T x) = \sum_{i=1}^{n} x_i(\log x_i - \log(1-b^T x))$$

$$= \left(\sum_{i=1}^{n} x_i \log x_i\right) - \left(\sum_{i=1}^{n} x_i\right) \log (1-b^T x)$$

and when performing these operations in this order we have the following operations:

$\log x_i$	log	1	1
$x_i \log x_i$	1	1	1
So $\sum x_i \log x_i$	n	n	n
$\sum x_i$	n	n	0
$\log(1-b^T x)$	log	2n	$3n^2$
$\left(\sum x_i\right)\log(1-b^T x)$	1	3n	$6n^2$
So the sum becomes	(n+1)log + 3n	9n	$9n^2 + 3n$
& the overall total	(n+1)log + $3n^2$	$4n^2 + 26n$	$31n^2$

Table 1.2. Revised operations count

We note that the n^3 term in the calculation of $\nabla^2 F$ no longer occurs and all three calculations are now dominated by order n^2 terms.

We may first conclude from this that the order in which a function calculation is performed can dramatically effect the cost of

calculating the derivatives by sparse doublet and sparse triplet methods.

We may note too that as reported by Dixon, Maany and Mohseninia (1989) and shown below, the times taken by the ADA implementation confirmed the constant nature of the ratios of the number of operations required to evaluate f, ∇f and $\nabla^2 f$; the difference in the actual ratios obtained do however imply that the cost of the index operations in the link lists cannot be ignored.

Dimension	Sparse Doublet	Sparse Triplet
5	1.68	33
10	3.20	49
20	5.27	46
30	5.55	41
40	5.69	46
50	5.95	48
60	6.23	48
200	7.23	
500	8.15	

Table 2. The relative cost of calculating ∇f by sparse doublet and $\nabla^2 f$ by sparse triplet to the cost of calculating f.

M 4.2 Reverse Automatic Differentiation; In Griewank (1988) a proof is given that by using reverse automatic differentiation whenever f can be calculated in M elementary operations then ∇f can be calculated in less than 5M elementary operations whatever the dimension n.

Recalling the notation for calculating f given in (2.1), reverse

automatic differentiation is simply:

For $i = 1, \ldots, n$

$\bar{x}_i = 0$

For $i = n+1, \ldots, n+M$

$x_i = F_i(x_j, x_k) \qquad j,k < i$

$\bar{x}_i = 0$

$f = x_{n+M}$

$\bar{x}_{n+M} = 1.$

For $i = n+M, \ldots, n+1$

$$\bar{x}_j = \bar{x}_j + \frac{\partial F_i}{\partial x_j} \cdot \bar{x}_i$$

$$\bar{x}_k = \bar{x}_k + \frac{\partial F_i}{\partial x_k} \cdot \bar{x}_i$$

$\nabla f_i = \bar{x}_i . \qquad i = 1, \ldots, n.$

This reduction in the number of operations required to calculate ∇f in reverse rather than forward automatic differentiation is balanced by the need to store the computational graph so that it is available for the reverse sweep.

One of my colleagues, Bruce Christianson (1990) has written an implementation of this in ADA as a standard "subroutine" compatible with our ADA optimisation codes. His results for the Helmholtz energy function are given in the following table 3. The method can be readily extended to obtain the product $\nabla^2 f.u$ for any given direction and these results are also given

Helmholz Energy Function	f ordinary	f by graph	∇f by graph	$\nabla^2 fu$ by graph
n = 20	-	7	8	16
50	11	53	52	120
100	40	320	300	660

Times in Seconds

Table 3 Reverse Automatic Differentiation

2.2 Jacobian of a vector function s_k (x)

As well as needing to calculate the gradient and Hessian of a scalar function f(x) we often need to calculate the Jacobian of a vector function s_k. This occurs in constrained optimisation where s_k could be the vector of active constraints, in ordinary differential equations or in the least squares minimisation of

$$f(x) = \sum_{k=1}^{k} s_k{}^2(x).$$

When each s_k is a separate calculation then we can use either sparse doublet or reverse automatic differentiation to efficiently calculate the gradient of each s_k. The sparse doublet form is particularly effective when each s_k only depends on a few variables (i.e. the Jacobian is sparse).

This is the case for the Lokta-Volterra predator prey function described in Byrne and Hindmarsh (1987).

Using the sparse doublet implementation in ADA, Parkhurst obtained the following results:

$$r_1 = \frac{\text{Evaluation of } (\underline{f}, J) \text{ using Sparse Doublets}}{\text{Evaluation of } (\underline{f}) \text{ using ordinary arithmetic}}$$

$$r_2 = \frac{\text{Evaluation of } (\underline{f}) \text{ using Sparse Doublets}}{\text{Evaluation of } (\underline{f}) \text{ using ordinary arithmetic}}$$

Dimension	50	200	800	1800	3200	5000
r_1	52.39	54.10	51.14	50.42	49.67	50.19
r_2	5.05	4.74	4.73	4.75	4.72	4.77

Table 4. Performance of Sparse Doublet Automatic Differentiation

Preliminary results with Christianson's implementation of reverse automatic differentiation indicate that this implementation gives a value of r_1 of approximately 19. If we assume that the s_k are separate calculations and that the total number of operations in calculating s_k is M_k, then the total operations for calculating ∇s_k is qM_k (where q is theoretically less than 5 for reverse automatic differentiation).

Now as $M = \sum M_k$ then the total number of operations for $J < \sum qM_k = qM$.

However, assuming that the s_k are separate calculations is not always fair as frequently there will be common strands in the calculations. If we drew a computational tree for the calculation of s_k such common strands must commence at the start of the tree and will probably involve a subset NEL_c of the variables and lead into a subset K_c of the subfunctions and will involve M_c elementary operations. Let us assume the common strand ends with a value x_c. There can of course be more than one common strand.

In sparse doublet arithmetic we would proceed forward through this common strand once and would use less than $q \times NEL_c \times M_c$ operations to obtain ∇x_c.

In reverse automatic differentiation we would reverse through the common strand K_c times and would thus appear to possibly need $q \times K_c \times M_c$ operations. This destroys the simple bound relating the operations involved in the calculation of $\underline{s}$ with those required by the calculation of J.

However suppose now we identified the common strand, and it would have to be identified if it is to be utilised in the calculation of $\underline{s}$, then we could start reverse differentiation at the end of the strand and calculate ∇x_c by one pass in q M_c operations.

Now suppose we calculate back in reverse mode for the gradient of s_k we will reach a value i=c and have $\bar{x}_c = \frac{\partial s_k}{\partial x_c}$ and can update those $\bar{x}_i$ i<n contained in NEL_c by $\bar{x}_i = \bar{x}_i + (\nabla x_c)_i \bar{x}_c$.

This linking operation involves $(NEL_c)(K_c)$ elementary operations, so we have replaced q $K_c M_c$ operations by $qM_c + (NEL_c)(K_c)$ operations. There is therefore a saving provided

$$q(K_c - 1)M_c > (NEL_c)(K_c)$$

which will frequently be true.

Similar considerations apply in the calculation of the Hessian matrix of a function. If we consider the calculation of the gradient of Rosenbrock's function by reverse differentiation it is simply

$\bar{x}_9 = 1$

$\bar{x}_8 = \bar{x}_9 \qquad \bar{x}_6 = \bar{x}_9$

$\bar{x}_7 = 2x_7\bar{x}_8$

$\bar{x}_1 = -\bar{x}_7$

$\bar{x}_5 = 100\bar{x}_6$

$\bar{x}_4 = 2x_4\bar{x}_5$

$\bar{x}_3 = \bar{x}_4 \qquad \bar{x}_2 = -\bar{x}_4$

$\bar{x}_1 = \bar{x}_1 + 2x_1\bar{x}_3$

The last step illustrates that not all the operations generated by reverse differentiation are elementary and we strictly need to divide this into two steps

$$\bar{x}_{1\frac{1}{4}} = 2x_1\bar{x}_3$$

$$\bar{x}_1 = \bar{x}_1 + \bar{x}_{1\frac{1}{4}}$$

The combined tree for the function and gradient calculation is therefore that given on the left

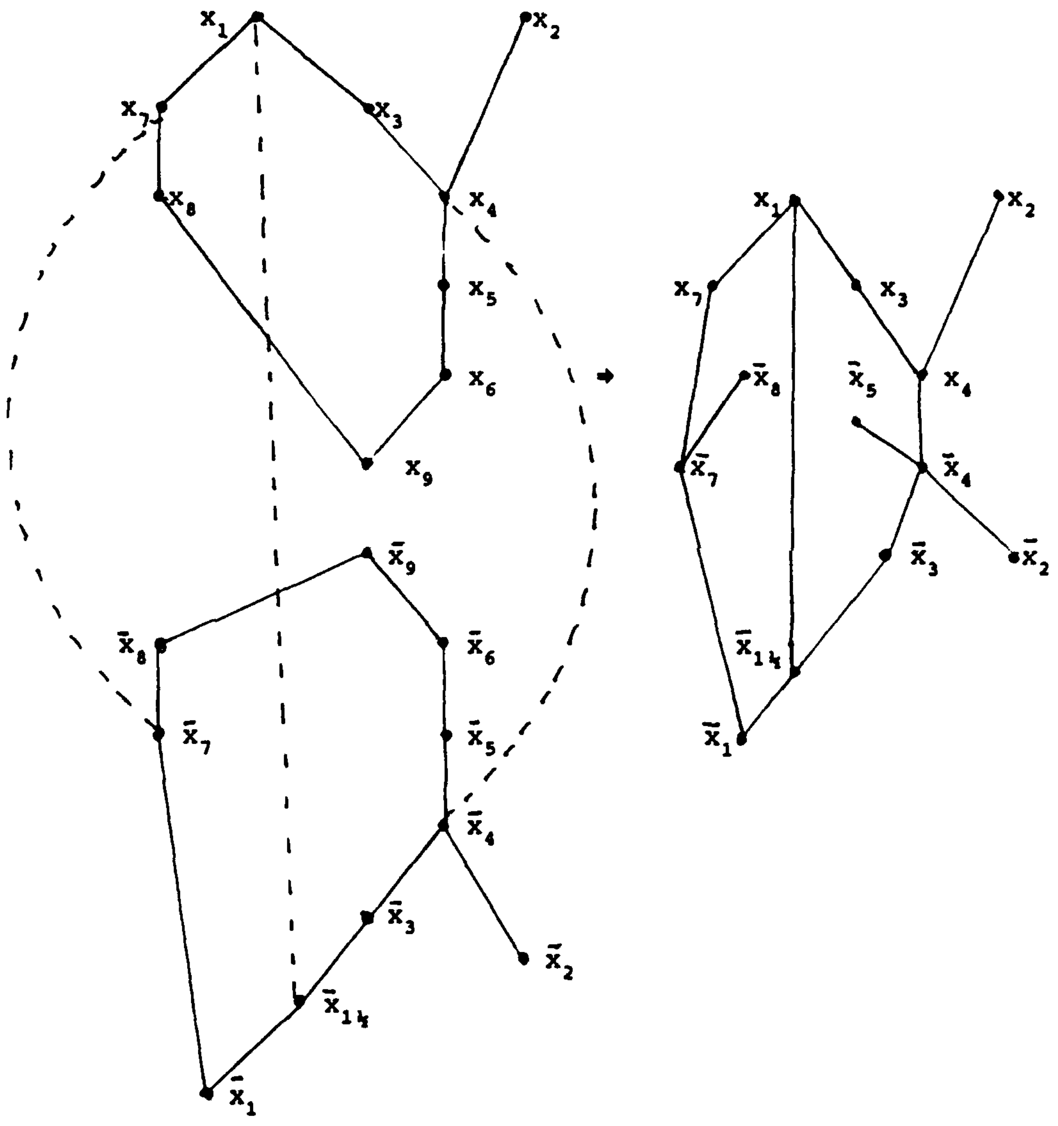


Figure 3

We may make the output a scalar, the directional derivative

$\nabla f^T d = \bar{x}_1 d_1 + \bar{x}_2 d_2$

by adding an extra node at the bottom and then apply reverse

differentiation again to obtain $\nabla^2 fd$. However we now need to reverse back from $\bar{x}_1$, $\bar{x}_2$ to x_1, x_2 and we note that this can only be done through the dotted lines linking the two graphs i.e. the nonlinear operations. This observation allows us to drastically reduce the tree as shown on the right and even now we have the common subtree in the calculation of $\bar{x}_2$, $\bar{x}_1$ namely that from x_1, $x_2 \rightarrow \bar{x}_4$. We may note too that the graph is symmetric about a horizontal line passing through the nonlinearities. This is a general result for all such "Hessian" graphs. Whether this approach to the Hessian calculation has any advantage over that described and implemented by Christianson still needs to be determined.

2.3 Conclusions

In this section we have demonstrated that two automatic methods now enable us to compute partial derivatives efficiently. In particular the sparse doublet/sparse triplet approach gives efficient methods for calculating the gradient and Hessian of partially separable scalar functions and of the Jacobian of vector functions.

The reverse differentiation approach requires the storage of the computational tree but then provides a very efficient way of calculating the gradient of any scalar function and the Jacobian of vector functions that do not contain significant subtrees. The method also provides a very efficient means of calculating $\nabla^2 f\, u$ for given u.

A method of handling common subtrees in reverse differentiation has been suggested above which should be used whenever $qM_c > NEL_c$.

Some interesting conjectures have been made on the reduction of the computational trees for Hessian calculations.

The contrasting values for sparse forward and backward calculation are

	Reverse	Sparse Forward
$\nabla f/f$	q	NEL
$\nabla^2 f/f$	q^2	½NEL(NEL+1)
J/f	qk*	NEL
$\nabla^2 fu/f$	2q (ref[13])	2.NEL.
	$q < 5$	

where NEL is the average number of nonzeros in the gradient vector ∇x_i $i > n$ during the calculation, if the function is partially separable NEL is less than the number of variables in each subfunction.

* We have however shown that a better bound for J/f by reverse automatic differentiation exists.

3. The effect of Parallel Computation

3.1 Experience using the ICL/DAP

At Hatfield Polytechnic our investigation into the benefits of using parallel computation commenced with the study of Kanu Patel who implemented unconstrained optimisation algorithms on two very different parallel processing machines - the Neptune machine at Loughborough University and the ICL-DAP at Queen Mary College. The ICL-DAP was an SIMD machine with 4096 processors, the Neptune an MIMD machine with four processors. His work on the Neptune was mainly concerned with the solution of small dimensional multi-extremal problems and is not particularly relevant to this paper.

As the DAP is an SIMD machine, each of the 4096 processors must either perform an identical arithmetic operation or be turned off. On this basis he chose to perform function evaluations in parallel and to then approximate the gradient and Hessian by difference formula, and thus implemented a parallel modified Newton algorithm, Patel (1982).

3.1.1 The Modified Newton Algorithm

Patel's Parallel Modified Newton Algorithm

Step 1 Initial guess $x^{(0)}$, step h, tolerance ε

Step 2 Calculate $f(x \pm ha_i \pm ha_j)$ all $j > i$ where a_i is a unit vector along i^{th} axis. This is an obvious parallel calculation.

Step 3 Calculate ∇f and $\nabla^2 f$ by central differences. He showed this could be performed in parallel on an SIMD machine.

Step 4 Solve the set of linear equations

$$(\nabla^2 f + \mu I)p = -\nabla f \tag{3.1}$$

He used the standard DAP equation solver.

Step 5 Perform a 4096 grid search in the two dimensional space spanned by p and $-\nabla f$ which sensibly replaces the line search given that 4096 evaluations take the same time as 1.

Step 6 $$x^{(k+1)} = \text{Arg min } f(x_i) \qquad i = 1, \ldots, 4096$$

using the very fast DAP "min" operation

Step 7 Return to 2 with $k = k+1$.

Essentially this algorithm uses $P = n^2$ processors and calculated f, ∇f, $\nabla^2 f$ in M steps and solved the Newton equations in $O(n^3/P)$ steps.

At the time when the relative speed of calculating 4096 parallel function evaluations on the DAP was 20 times faster than evaluating them sequentially on a DEC 1091, the comparison times to solve five 64 dimensional problems were obtained. Standard Modified Newton, Variable metric and Conjugate gradient codes were run on the DEC 1091 and the parallel Newton on the DAP.

Problem	Sequential Newton	Variable Metric	Conjugate Gradient	Parallel Newton
Quadratic	16.60	2.10	1.18	0.966
Extended Rosenbrock	140.98	80.78	11.36	36.626
Extended Powell	134.97	41.28	11.06	1.500
Trigonometric	2604	455	78.68	17.57
Extended Box	7196	1263	354.0	202.37

Table 5 Performance of the Parallel Newton Method

From these results it is easily seen that the parallel Newton algorithm greatly outperformed the sequential Newton code, often by far more than the natural value of 20. However, it was also obvious that on these functions the conjugate gradient code also outperformed the sequential Newton code and that a comparison of its times with those of the parallel code was not particularly encouraging.

Two decisions were made at that time which have greatly influenced our subsequent development. The first was that as the functions we were interested in at that time possessed considerable structure this should be used in the evaluation of f and in the solution of the set of equations. The second was that we needed to consider conjugate gradient type algorithms and problems with dimensions greater than 64.

3.1.2 The parallel conjugate gradient algorithm

The first problem studied was a quadratic function derived from Stones set of heat conduction problems namely:-

$$f = \iint K_x \left(\frac{\partial T}{\partial x}\right)^2 + K_y \left(\frac{\partial T}{\partial y}\right)^2 - 2QT \, dA \qquad (3.2)$$

To solve this problem Ducksbury (1984) split it into rectangular elements and approximated T in each element by the standard bi-linear mapping.

The function can now be expressed in the form

$$f = \sum_{\text{elements}}^{K} s_{\bullet 1}$$

where

$$s_{\bullet 1} = \iint_{\text{element}} K_x \left(\frac{\partial T}{\partial x}\right)^2 + K_y \left(\frac{\partial T}{\partial y}\right)^2 - 2QT \; dA \qquad (3.3)$$

and $s_{\bullet 1}$ is a function of 4 variables, the values of T at the corners of the elements.

If $M_{\bullet 1}$ is the number of operations needed to evaluate $s_{\bullet 1}$, then the total operation count for f is simply C $M_{\bullet 1}$ where C is the number of elements and is slightly less than n.

Again $$\nabla f = \sum_{el} \nabla s_{\bullet 1}$$

$$\nabla^2 f = \sum_{el} \nabla^2 s_{\bullet 1}$$

where $\nabla s_{\bullet 1}$ only has four nonzeros and $\nabla^2 s_{\bullet 1}$ 16 nonzeros.

The resulting set of linear equations was solved on the DAP using the conjugate gradient algorithm. The decision was taken to allocate each element to its natural processor on the DAP grid and to distribute the search direction p over the processors so each processor held the four components that influence its value of f. Given the conjugate gradient formula this only required data passing between neighbouring processors thus reducing the communication costs, the conjugate gradient code then only needs the calculation of

$$y = \nabla^2 f \; p = \sum_{el} \nabla^2 s_{\bullet 1} \; p_{\bullet 1} = \sum_{el} y_{\bullet 1} \qquad (3.4)$$

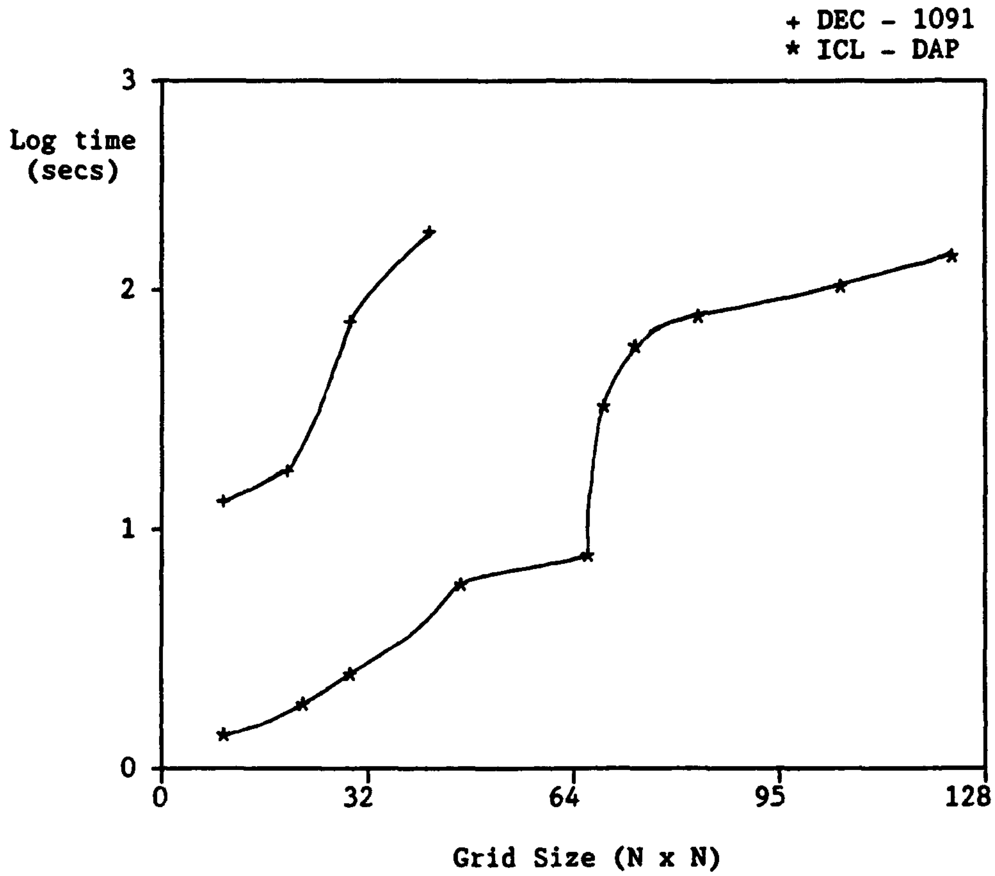


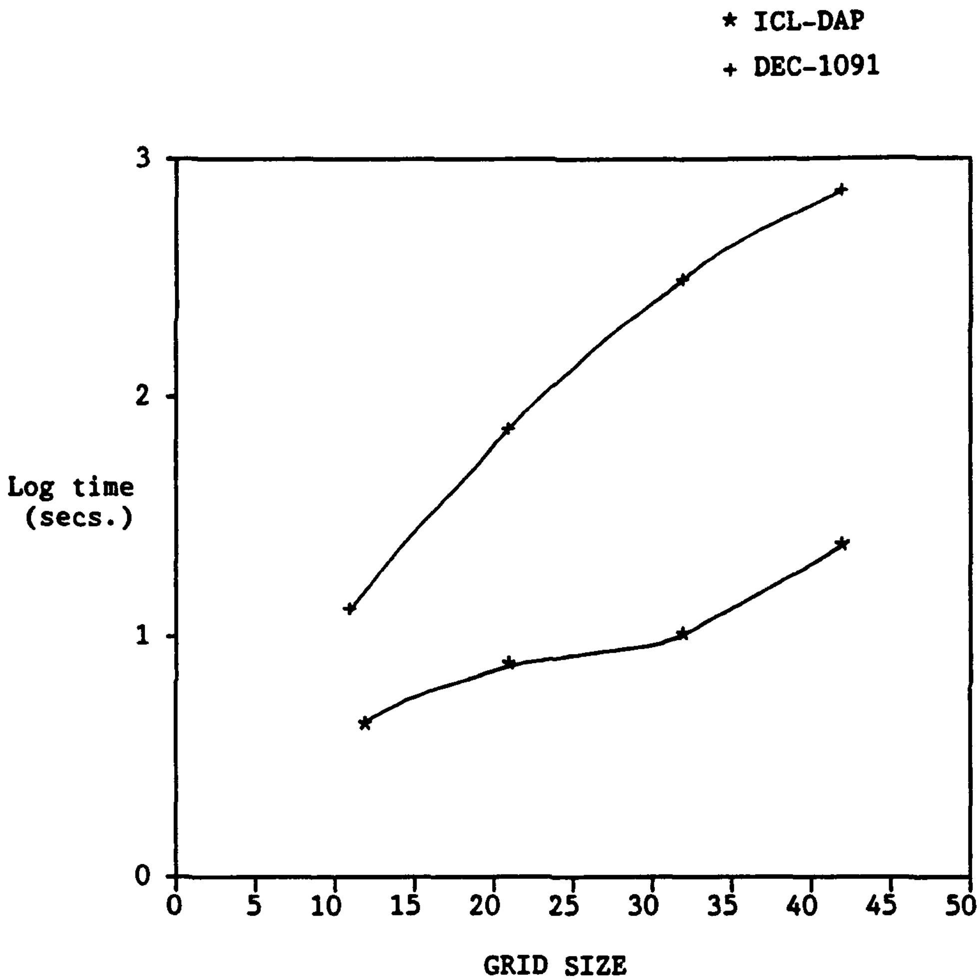
Figure 4.1 Performance of the Conjugate Gradient Codes

and for each element/processor this is simply 16 operations. In practice the scalar products $r^T r$, $y^T p$, $p^T p$ were also calculated using partial vectors distributed over the processors.

This problem is equivalent to the heat conduction problem tested by Stone who introduced a number of different cases based on the distribution of K_x and K_y.

The simplest case with $K_x = K_y = 1$ leads to a well conditioned problem and the performance of the conjugate gradient codes on this problem are shown in Figure 4.1, where the log of the CPU time is plotted against the grid size.

Other distributions lead to less well-conditioned problems but the relative performance of the sequential and parallel codes is unaltered as shown in Figure 4.2.



<u>Figure 4.2 Performance of Sequential and Parallel Codes</u>

The time required on the sequential machine precluded running the test for all the grid sizes, but there was a speed-up of over 100 even when only using a small part of the DAP.

This work was extended to the nonlinear domain by considering the differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - Ru\,\frac{\partial u}{\partial x} = 0 \tag{3.5}$$

with boundary values such that the analytic solution was

$$u = \frac{6x_1}{Rx_2{}^2}$$

and converting it to a three dimensional map by introducing

$$a = \frac{\partial u}{\partial x}\,,\ b = \frac{\partial u}{\partial y}$$

and minimising

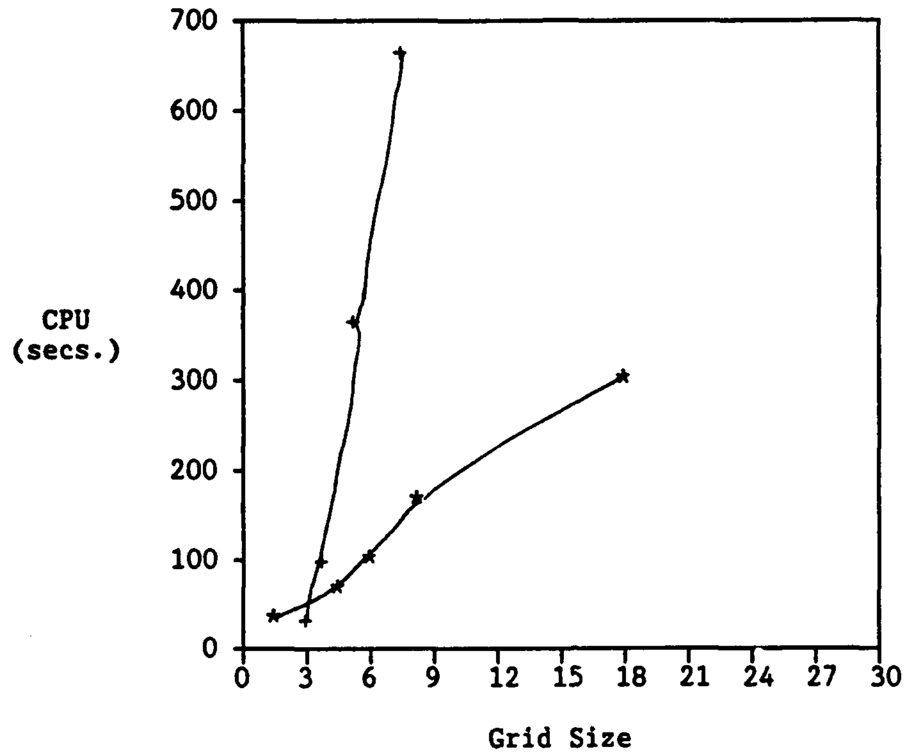
$$f = \iint \left(a - \frac{\partial u}{\partial x}\right)^2 + \left(b - \frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial a}{\partial x} + \frac{\partial b}{\partial y} - Rua\right)^2 dA \tag{3.6}$$

Again the function and gradient evaluations were distributed over rectangular elements so that for a 64 x 64 grid there were 3 x 4096 unknowns before the specification of the boundary condition. This implies that the Hessian matrix is now a combination of 12 x 12 blocks. Again a parallel version of the Fletcher Reeves algorithm was used.

A typical result is shown in figure 5(overpage), where the number of processers used P = (grid size)^2. It appears that the function/gradient cost is proportional to the (grid size)^2 sequentially and is therefore constant for the parallel machine and that the number of iterations is approximately linear in the grid size.

So the sequential CPU α (grid size)^3

parallel CPU α (grid size).



* ICL DAP
+ DEC 1091

Figure 5 Performance of Parallel and Sequence Conjugate Gradient Codes

3.1.3 The Truncated Nevton Algorithm

Before 1983 most existing optimisation routines that used the Nevtons equation

$$r = \nabla^2 f(x^k)d + \nabla f(x^k) = 0$$

solved the equation as accurately as possible even though the current iterate vas far from the optimal solution. For large problems a major part of the computing cost can be attributed to the solution of these equations.

The fact that it is not necessary to solve these equations accurately when far from the solution, was reported by Dembo, Eisenstat and Steihaug (1982, 83). They introduced the idea of calculating an approximate solution when far from the solution and suggested solvingthe equations by the conjugate gradient method and also truncating the iteration before an accurate solution is obtained.

In our implementations at Hatfield this inner conjugate gradient iteration was terminated if

(i) $r_j^T r_j < \nabla f^T \nabla f * \min\left(\frac{.1}{k^2}, \nabla f^T \nabla f\right)$

where ∇f is calculated at the k^{th} iterate of the outer iteration and r_j at the j^{th} iterate of the inner iteration or

(ii) $d_{j+1}^T d_{j+1} > D^2$

a trust region imposed to limit the size of the step taken in the inner iteration

The conjugate gradient method has the property that

$$d_{j+1}^T d_{j+1} > d_j^T d_j$$

so this limit will be reached and terminate the inner iteration unless the solution is within the trust region

<u>or</u>;

(iii) $d_j^T \nabla f > -.1||d_j|| \; ||\nabla f||$

which ensures that the resultant direction satisfies Wolfes first condition for finite termination.

These conditions when combined with an Armijo style line search that satisfies Wolfes Conditions II and III ensure that the truncated Newton method has finite termination to the neighbourhood of a stationary point.

Code	Problem	N	No.F	No.G	EFE's	C.P.U.
TRUNEW	Wood	4	103	229	1019	0.48
OPCG	Wood	4	384	155	1004	0.80
OPVM	Wood	4	128	77	436	0.90
E04KDF	Wood	4	54	202	862	0.42
TRUNEW	Ext. Powell	4	39	82	367	0.28
OPCG	Ext. Powell	4	96	40	256	0.45
OPVM	Ext. Powell	4	9	31	173	0.46
E04KDF	Ext. Powell	4	18	90	378	0.31
TRUNEW	Ext. Powell	60	33	90	5433	1.27
OPCG	Ext. Powell	60	117	48	2997	1.87
OPVM	Ext. Powell	60	341	194	11981	59.85
E04KDF	Ext. Powell	60	42	1302	78162	15.31
TRUNEW	Ext. Powell	80	31	95	7631	1.81
OPCG	Ext. Powell	80	216	93	7656	3.79
OPVM	Ext. Powell	80	439	251	20519	134.71
E04KDF	Ext. Powell	80	43	1643	131483	32.84
TRUNEW	Ext. Rosenbrock	10	41	59	631	0.34
OPCG	Ext. Rosenbrock	10	99	33	429	0.54
OPVM	Ext. Rosenbrock	10	133	83	963	2.00
E04KDF	Ext. Rosenbrock	10	41	181	1851	0.53
TRUNEW	Ext. Rosenbrock	20	49	71	1469	0.56
OPCG	Ext. Rosenbrock	20	139	40	939	1.00
OPVM	Ext. Rosenbrock	20	198	113	2447	6.27
E04KDF	Ext. Rosenbrock	20	52	312	6292	1.21
TRUNEW	Ext. Dixon	80	21	148	11861	2.16
OPCG	Ext. Dixon	80	136	87	7146	3.04
OPVM	Ext. Dixon	80	201	102	8361	31.14
E04KDF	Ext. Dixon	80	28	929	66268	13.54
TRUNEW	Ext. Powell	2000	41	109	218041	24.98
OPCG	Ext. Powell	2000	125	49	98125	50.13
TRUNEW	Ext. Dixon	2000	29	851	1702029	206.42
OPCG	Ext. Dixon	2000	1064	520	1041064	393.47

Table 6 Comparison of the truncated Newton code (TRUNEW) with a conjugate gradient (OPCG), variable metric (OPVM) and modified Newton code (E04KDF).

Our first results comparing this algorithm with the standard optimisation codes were reported in Dixon & Price (1986, 88) and confirmed the evidence reported by Dembo, Eisenstat and Steihaug that

this method was much more efficient than Modified Newton, Variable Metric or Conjugate Gradient codes over a wide range of dimensions. For convenience these results are shown in Table 6.

In this code the matrix vector product

$$\nabla^2 fp$$

required in the truncated Newton method, was obtained by a difference approximation

$$\nabla^2 fp = (\nabla f(x+hp) - \nabla f(x))/h$$

as suggested by Dembo et al.

Ducksbury (1984) then implemented a parallel version of the Dembo-Steihaug Truncated Newton algorithm on the DAP and compared the relative performance of the parallel conjugate gradient algorithm with the parallel truncated Newton method on a large number of problems including ones based on the partial differential equation (3.5).

He noted that the Truncated Newton method consistently outperformed the conjugate gradient algorithm.

A typical result is one in which the sequential conjugate gradient code required in excess of one hour to solve a 39 x 39 grid (4111 unknowns) and the parallel conjugate gradient code on the DAP just 34 secs, this being a speed up of 104 over the DEC 1091 at a point where only 3/8 of the processors of the DAP were in use.

For a finer grid of 64 x 64 processors (12036 unknowns) the conjugate gradient code on the DAP required 50.76 secs while the truncated Newton method only required 13.21 secs.

Turning now to the Navier Stokes problem,

$$u\frac{\partial u}{\partial x_1} + v\frac{\partial u}{\partial x_2} = -\frac{\partial p}{\partial x_1} + \frac{1}{R}\nabla u^2$$

$$u\frac{\partial v}{\partial x_1} + v\frac{\partial v}{\partial x_2} = -\frac{\partial p}{\partial x_2} + \frac{1}{R}\nabla^2 v$$

$$\frac{\partial u}{\partial x_1} + \frac{\partial v}{\partial x_2} = 0$$

This was converted into a set of 6 first order equations by introducing the fields

$$a = u \ ; \quad b = \frac{\partial u}{\partial x_1} \ ; \ c = \frac{\partial u}{\partial x_2}$$

$$d = v \ ; \quad e = \frac{\partial v}{\partial x_1} \ ; \ f = \frac{\partial v}{\partial x_2} \text{ and } p$$

and noting that as the continuity equation is simply $b + f = 0$ the variable f can be eliminated.

The equations become

$$e_1 = ab + cd - \frac{1}{R}\left(\frac{\partial b}{\partial x_1} + \frac{\partial c}{\partial x_2}\right) + \frac{\partial p}{\partial x_1}$$

$$e_2 = ae + df - \frac{1}{R}\left(\frac{\partial b}{\partial x_1} + \frac{\partial c}{\partial x_2}\right) + \frac{\partial p}{\partial x_2}$$

$$e_3 = b + f$$

$$e_4 = \frac{\partial a}{\partial x_1} - b$$

$$e_5 = \frac{\partial a}{\partial x_2} - c$$

$$e_6 = \frac{\partial d}{\partial x_1} - e$$

$$e_7 = \frac{\partial d}{\partial x_2} - f$$

and the objective function is

$$I = \text{Min} \int \sum_1^7 \ e_k^{\ 2} dA$$

The minimization is carried out over the field variables and as the integrand only contain first derivatives the theory of the calculus of variables provides natural boundary conditions that will be satisfied on the boundary if any of the seven fields are not fully specified.

As we now have seven unknowns at each node point the storage requirement for $\nabla^2 I_{\bullet 1}$ for each finite element increases to 28 x 28 for the truncated Newton method and this was not available on the DAP when these tests were run.

	3 x 3	5 x 3	9 x 9	17 x 17
Value of I	0.2037	0.080917	0.03533	0.01730
CPU	2:56	5:39	14:11	13:51
Total CPU		36:37		
Iterations	82	164	417	389

Table 7 *Performance on the Navier Stokes Equation*

3.2 More thoughts on calculating the search direction

3.2.1 The Truncated Newton Method with Automatic Differentiation

Our first results obtained combining the concept of automatic differentiation with the truncated Newton algorithm used a crude Fortran implementation and were reported in Dixon & Price (1986,89).

In that paper it was noted that the vector matrix product $\nabla^2 f\ p$ could be obtained in two ways,

(1) Sparse triplets

First form $\nabla^2 f$ and then form the sparse matrix vector product $\nabla^2 fp$

(2) Structured triplets

Modify the definition of a triplet to be

$$(f, \nabla f, \nabla^2 f p)$$

and alter the basic algebra appropriately.

It is important to notice that a structured triplet simply consists of two sparse vectors and a scalar and therefore requires far less store. Mohseninia implemented both versions in ADA and found that the structured version is much slower.

Problem	Dimension	Structured Sparse Triplets	Sparse Triplets
Extended Rosenbrock	2	.81	.62
	20	41.12	4.48
	40	98.61	11.66
	80	420.08	39.64
Extended Powell	4	.75	.57
	20	27.89	3.36
	40	112.3	9.02
	80	760.7	33.42
Extended Dixon	2	.19	.13
	20	26.06	2.96
	40	103.51	12.6
	80	546.13	42.31

Table 8 Comparison of structured and sparse triplet automatic differentiation.

We have therefore not pursued the concept of structured automatic differentiation further. It does however require far less store when this is important.

3.2.2 Accurate Arithmetic

While these results were fairly conclusive in implying that sparse triplets were preferable to structured triplets. They were worrying in that they implied that the codes were not behaving theoretically correctly on these extended functions.

These functions have the property that as their dimensions increase the number of distinct eigenvalues remain constant.

In 1974 Dixon had conjectured that for such extended problems the number of iterations of variable metric and conjugate gradient algorithms should be independent of n.

This was proved in Spedicato (1976)

It was however clearly not occurring in the above results. The reason appears to be that while theoretically the value of the scalar product

$$\sum_{i=1}^{n} a_i b_i$$

is independent of the ordering of the elements in most computer languages this is not true and this breaks the symmetry of the calculations in the variable metric algorithm.

Dave Mills (1990) wrote a simple sort to make the scalar product independent of the order and probably more accurate and obtained results that reflected the theoretical result.

A typical result obtained using the variable metric algorithm on the extended Powell problem is given overpage.

Dimension	Number of iterations using double precision arithmetic	Number of iterations using sorted dot products
4	39	39
8	64	39
16	25	39
32	67	39
64	52	39
128	165	39

Table 9. The effect of accurate arithmetic

Due to the result of this experience I would urge people to be very careful how they calculate scalar vector products, and matrix vector products to ensure that the arithmetic does not destroy the symmetric structure of the problem.

It is essential in the design of BLAS software for general purpose use that accuracy is not sacrificed for speed.

3.2.3 Maany's Test set

To overcome the problems with the extended functions so often used to test algorithms in large dimensions Maany (1989) (TR210) introduced a new family of test problems.

$$f(x) = 1.0 + \sum_{i=1}^{n} 0.5\left(\frac{i}{n}\right)^{k} x_i^{\,2}$$

$$+ \sum_{i=1}^{n-1} \beta(x_i x_{i+1} + x_i x_{i+1}^{\,2})^2 + \sum_{i=1}^{2N} \delta\, x_i^{\,2} x_{i+N}^{\,4}$$

$$+ \sum_{i=1}^{N} \delta\left(\frac{i}{n}\right)^{K} x_i x_{i+2N}$$

Here the dimension $n = 3N$, and the family contains three parameters β, δ, K.

With $K = 0$ the eigenvalue pattern is similar to that in an extended family but for other values of K this property is lost and the family gets more ill-conditioned as K increases.

If $\beta = \delta = 0$ the problem is quadratic and diagonal, when $\beta \neq 0$ the diagonals on either side of the main diagonal are introduced and $\delta \neq$ introduces four wide diagonals.

Maany tested the truncated Newton code including sparse triplet automatic differential on twelve cases drawn from this family for a range of dimensions up to $n = 3000$.

His results indicated the robustness of the truncated Newton code and the efficiency of the dynamic data structure used in ADA.

Three sets of his results on badly conditioned problems are detailed below:

No Preconditioning $n = 3000$						
	β	δ	K	No. of iter.	CPU time	% in AD
Case 10	1/16	1/16	2	8498	19001	12
Case 11	1/8	1/8	2	6482	15461	15
Case 12	0.26	0.26	2	17211	38072	11

Table 10.1. The truncated Newton Method and automatic differentiation on the Maany problem.

These results indicated that the percentage of time being spent in conjugate gradient code dominated that spent in automatic differentiation.

The results were therefore repeated using a diagonal pre-conditioner.

Diagonal Preconditioning $n = 3000$						
	β	δ	K	No. of iter.	CPU time	% in AD
Case 10	1/16	1/16	2	31	1220	84
Case 11	1/8	1/8	2	31	1226	84
Case 12	0.26	0.26	2	31	1318	84

Table 10.2. The truncated Newton Method and Automatic Differentiation on the Maany problem. Effect of preconditioning.

These results indicated that it was essential to use a preconditioner but that when doing so the calculation of the gradients even with a sparse automatic differentiation routine could dominate. We therefore wished to go on a parallel computing system to see how these results would improve.

Before doing so however it is appropriate to mention the results of Vespucci (1990) who has shown that if the conjugate gradient algorithm was replaced by that equivalent algorithm from the Abaffy-Broyden-Spedicato family which theoretically generates the same sequence of points x^k, then the number of outer and inner iterations is greatly reduced.

Two typical results one for a 40 dimensional modified extended Wood function with a complete set of distinct eigenvalues and one for a member of the Maany family with $n = 150$ and $K = 3$ indicate this effect.

		Maj. It.	Min. It.	Time
Modified Extended Wood $n = 40$	CG ABS/CG	412 137	14962 3218	16.1 7.5
Maany Function $n = 150$	CG ABS/CG	222 32	30082 1519	400 47

Table 11. The effect of using the conjugate gradient algorithm from the ABS family on iterations and computer time

These results emphasise the need for accurate arithmetic in CG algorithms. The results are not strictly comparable as the ABS/CG method stores a full set of n x n matrices H_i and it is indeed remarkable that the cost of updating such a matrix is dominated by the effect of rounding error in increasing the number of major and minor iterations.

In her paper she advocates replacing the conjugate gradient algorithm by a truncated LL^T iteration implemented within the ABS class but using the sparsity of matrix H_i.

A few of her results with this code are given in the following table:

Modified Extended Wood				
n = 40	CG	412	14962	16.1
	ABS/CG	137	3218	7.5
	ABS/LL^T	73	1967	1.8
n = 100	CG failed to converge in 1000 it.			
	ABS/CG	303	11704	161
	ABS/LL^T	130	7917	36
Maany Function				
n = 150 K = 3	CG	222	30082	300
	ABS/CG	32	1519	47
	ABS/LL^T	51	1651	15
n = 300 K = 3	CG	198	52601	115
	ABS/LL^T	75	6405	35
n = 900 K = 3	CG	138	116620	595
	ABS/LL^T	83	8149	41

Table 12 Use of the ABS/LL^T code

Because of the sparsity of H_i for the LL^T algorithm no difficulty was experienced in storing the matrix and the effectiveness of the new approach is obvious.

Analysis of the results does however indicate that if the use of accurate arithmetic reduced the number of iterations used by the CG algorithm to the number required by the ABS/CG algorithm then the time needed by the CG algorithm would be lowest.

3.3 Results using the Sequent Balance System

To obtain results using automatic differentiation on a parallel processer we needed to have access to a parallel machine that could run ADA and supported concurrent tasks. The Sequent Balance fulfilled this criteria and Professor Delves allowed us access to his machine at Liverpool University.

Mohseninia (1989) implemented automatic differentiation (sparse triplets) and the truncated Newton method on this system. As the Sequent Balance contains far fewer processors than the DAP a number of elements needed to be allocated to each processor. Again as the processors are not allocated so that only nearest neighbour communication is possible, the effect of data communication is more obvious.

Typical times for the sparse triplet evaluation of f, ∇f, $\nabla^2 f$ on the Sequent Mohseninia (1989) are shown below for the Olsen square cavity problem.

Elements	Number of Processors Used									
	1	2	3	4	5	6	7	8	9	10
8	37	20	12	10	11	12	13	8	-	-
32	156	85	62	44	38	35	26	22	24	26
128	670	352	227	175	146	124	107	92	87	81

Table 13 Parallel Sparse Automatic Differentiation

These results were really very satisfactory and indicated that dividing automatic sparse triplet differentiation into concurrent tasks is effective.

The effect of data communication becomes even more obvious however when considering the operations required within the truncated Newton code. The dominant operation of this part of the computation is the product

of the sparse matrix $\nabla^2 f$ with the search direction p, and the cost of this multiplication for dimensions 512 and 1107 within the cavity driven flow problem on the Sequent Balance were

Time	Number of Processors									
	1	2	3	4	5	6	7	8	9	10
n=512	3.80	2.40	2.20	2.10	2.08	2.10	2.11	2.12	2.2	2.4
1107	9.50	6.00	5.00	4.5	4.5	4.3	4.1	4.4	4.4	5.2

<u>Table 14 Parallel Matrix Vector Multiplication</u>

These results are really quite disappointing and indicate that communication costs cannot be ignored when performing linear algebra on parallel systems.

However calculating $\nabla^2 fp$ *is much less expensive* than forming $\nabla^2 f$ and using the figures given in the previous section and assuming P = 8 we have as an overall effect

	P = 1	P = 8
Time in AD on sequential machine	.84	.12
Time in linear algebra	.14	.07
Other time	.02	.02
	1.00	.21

Giving an overall speed up of approximately 5.

3.4 Results using a Transputer Net

At about this time the Polytechnic took delivery of a small transputer network and we began tests to discover its capabilities and to construct a model of its behaviour.

As the parallel linear algebra had proved disappointing on the Sequent, Jha began his investigation in this area and demonstrated that the operation Av could not be performed effectively in parallel if A had to be downloaded into the system (see table 15 below). In this table the measured times are given on 1, 2, 4 and 8 transputers and the expected time on 8 transputers using Jha's model of the communication time and computation time of the network.

Dim.	Number of Transputers							
	1		2		4		8	
	Act. Time	Exp. Time	Act. Time	Exp. Time	Act. Time	Exp. Time	Act. Time	Exp. Time
16	1	-	2	-	3	-	4	-
32	5	-	7	-	14	-	17	-
64	22	-	30	-	54	-	66	-
83	34	-	46	-	84	-	103	-
96	50	-	67	-	120	-	147	-
128	88	89.6	119	118.49	213	206.06	255	233.01

Table 15 REAL 64 Matrix-vector Multiplication

Calculation of expected timings in milliseconds

However, when the matrix vector multiplication was embedded within Jacobi's iterative algorithm

$$x^{K+1} = (I - A)\, x^{K} + b \qquad A_{ii} = 1$$

An effective speed up was obtained (see table 16).

Dim	Number of Transputers							
	1		2		4		8	
	No. of itr.	Time	Time	s.up	Time	s.up	Time	s.up
16	36	49	52	0.94	30	1.63	25	1.96
32	39	193	197	0.98	109	1.77	75	2.57
64	43	819	625	1.31	431	1.90	262	3.13
80	42	1236	832	1.46	610	2.03	384	3.22
96	42	1768	1097	1.61	760	2.33	549	3.23
128	46	3904	2039	1.92	1248	3.13	839	4.66

Table 16. Parallel Jacobi's Method on Network of Transputers.

The difference being mainly due to the fact that A only had to be down loaded once. The results obtained were again in agreement with the constructed model, it should perhaps be stressed that the model predictions are dominated by the communication requirements and not by the computational time.

Dim	Number of transputers							
	1		2		4		8	
	Time	Model	Time	Model	Time	Model	Time	Model
128	3904	3894	2039	2096	1248	1220	839	834

Table 17.

Similar good speed ups were obtained when a parallel conjugate gradient code was applied to a full matrix with distinct eigenvalues (see table 18), but no speed up could be obtained by implementing a sparse matrix code in parallel as the much smaller amount of computation at each iteration was dominated by the communication costs.(Table 19).

Matrix Size	Number of Transputers										
	1		2			3			8		
	Time	Itr	Time	s.up	itr	Time	s.up	itr	Time	s.up	itr
16	45	26	27	1.67	26	19	2.37	26	19	2.37	26
32	358	63	198	1.81	63	116	3.09	62	91	3.93	62
64	3416	167	2013	1.70	167	976	3.50	166	621	5.50	163
80	8937	253	4669	1.91	256	2438	3.67	279	1269	7.042	234
96	17786	355	9128	1.95	355	4242	4.10	347	2829	6.29	388
128	39461	449	21483	1.84	479	10754	3.67	513	6188	6.377	513

Full code version

For matrix A^TA

*** Times are in milliseconds**

Table 18. Parallel Conjugate Gradient

Dim	No. of NNZ	No. of iter.	Number of transputers				
			1	2		4	
			Time	Time	s. up	Time	s. up
16	74	15	21	22	0.95	25	0.84
32	154	25	70	69	1.01	75	0.93
64	314	38	212	202	1.05	214	0.99
80	394	43	300	283	1.06	298	1.01
96	474	47	394	369	1.07	387	1.02
128	634	55	614	572	1.07	599	1.07
IN FULL MATRIX CODE							
128	634	55	5131	2745	1.87	1498	3.43

Table 19. Sparse Conjugate Gradient with Matrix of distinct eigenvalues

Eigenvalue to 1, ..., n, so CN = N.
Sparse code.
* Time in milliseconds

From our experience we would stress the difficulty of getting speed up for the solution of sparse linear equations on parallel systems, and the necessity to model the communication time carefully when predicting the performance of parallel algorithms.

3.5 Concurrent Developments in Parallel Optimization

In this section an attempt will be made to put the methods described in the previous context into perspective alongside those undertaken elsewhere.

In one of the earliest reviews of the subject of parallel computing in optimization, Schnabel (1984), identified two classes of optimization problems that would benefit from parallel processing, namely,

(1) large problems with dimensions of more than 100

(2) the multi-extremal global optimization problem.

In this study we are concerned with the former problem.

Turning first to the classic situation of unconstrained optimisation in 1984. There were four established classes of sequential algorithms.

(1) Modified Newton methods in which the Hessian matrix G was calculated and then a modified Newton equation of the form

$$(G + \mu I)d = -g$$

solved by Choleski decomposition. If M was the number of steps needed to calculate the function value then each iteration in the days before automatic differentiation involved approximately

$$\tfrac{1}{2}(n)(n+1)M + 1/6n^3$$

operations.

(2) Variable metric methods in which either an approximation B to the Hessian matrix G was updated using one gradient evaluation or an approximation H of the inverse was used. As methods for updating the Choleski decomposition were known both versions could be performed in

$$(n+1)M + cn^2$$

operations.

As it was found that these methods rarely required n times the number of iterations than the modified Newton this was usually preferred if n > 5.

(3) Conjugate gradient methods of the Fletcher-Reeves (1963) or Polak Ribiere (1971) type that had the property that they would minimise a quadratic function in a number of iterations equal to the number of distinct eigenvalues.

At each iteration these methods only need

$$(n+1)M + c_2 n \text{ operations}$$

and became standard for $n > 100$.

(4) Direct search methods where the gradient or Hessian were not evaluated or estimated. The most successful of these was undoubtedly the Nelder and Mead (1965) Simplex algorithm which regrettably frequently converges to an incorrect point if $n > 5$.

(5) As mentioned earlier in this paper this situation was dramatically altered by the advent of the truncated Newton method which contained two types of iteration.

An outer iteration requiring

$$\tfrac{1}{2}n(n+1)M \text{ operations}$$

and a much more frequent inner iteration essentially dominated by NNZ operations where NNZ is the number of nonzero in G.

This method was usually more efficient than any of the others, but naturally each still has its advocates and parallel versions of each have been investigated.

For instance in Lootsma (1986) a similar approach to Patel's parallel modified Newton method is described, in Straetter (1973) a parallel variable metric algorithm is given, while many authors have noted that the vector operations in the conjugate gradient algorithms are ideally suited for implementation in parallel. Chazan and Miranker (1970) described an early parallel direct search method based on parallel directions, an idea extended in Sutti (1983).

The parallel modified Newton methods typically estimated the Hessian matrix by performing $\frac{1}{2}n(n+1)$ parallel function evaluations or n parallel gradient calculations so that the parallel operation count became

$$M + c_3 n^2 \text{ operations.}$$

This requirement for exactly $P = \frac{1}{2}n(n+1)$ or $P = n$ processors is however very restrictive so Byrd, Schnabel and Shultz (1988) consider methods that used parallel gradient evaluations with $2 < P < n$ or parallel function evaluations with $2n+1 \leq P \leq (u^2 + 3n)/2$. In their paper they argued in favour of using the unfactored inverse H form of the variable metric algorithm and divided its rows between the P processors, and either updated it P times per iteration for the P gradient calculations performed, or formed P correct columns of the matrix using this information; further details of this method are also given in Schnabel (1988).

Van Laarhoven (1985) applied Straetters ideas for parallel variable metric algorithms to the larger Huang class of updating formula. He showed that the rank one update is the only one which provides a parallel extension with the property of quadratic termination, in the sense that n updates using linearly independent directions d_i produce a matrix that is equal to the Hessian or its inverse. This approach was first tested on large problems by Dayde (1989), it was discussed further in Dayde, Lescrenier and Toint (1989) who conclude that Newton's method using finite differences of the gradient to estimate the Hessian is much more efficient.

The problem of designing a direct search method for use on parallel computers has been resolved by the research of Torczon (1989). She showed first that the Nelder & Mead algorithm can and does frequently fail on large dimensions, then showed how to design a sequential direct search method that theoretically and practically converges in large dimensions and finally constructed a parallel version that has a relative speed up as P increases (i.e. speed up > P). This result is of considerable theoretical importance. It is the only method that effectively utilises $P > n^2$ processors, but has the disadvantage that even for small n it is very much more expensive than parallel gradient methods. Further developments of this research are given in Dennis and Torczon (1990).

All the above research was carried out before automatic differentiation became available. Using sequential automatic differentiation the costs of all four gradient classes reduce.

		Before	With
Modified Newton		$\frac{1}{2}n(n+1)M + 1/6n^3$	$qnM + 1/6n^3$
Variable Metric		$(n+1)M + cn^2$	$qM + cn^2$
Conjugate gradient		$(n+1)M + c_2n$	$qM + c_2n$
Truncated Newton	outer	$\frac{1}{2}n(n+1)M$	qnM
	inner	NNZ	NNZ

Again our experience is that the truncated Newton method is by far the most effective. In this report we have described many implementations where the parallelism is used within the function evaluation effectively reducing M to

$$k\frac{M}{P},$$

the alternative (approach B) use of parallelism is to perform the n gradient calculations in parallel effectively reducing n to

$$\frac{n}{P}.$$

This has been implemented by Sofer and Nash (1990) who also give a new parallel equation solver thus reducing the cost of the inner iterations.

Both codes are very effective. Essentially P in the Sofer-Nash approach is limited to n and the Dixon-Maany-Mohseninia approach to K a number determined by the partially separable nature of the function.

However, the two parallelisms are complementary and if sufficient processors were available on a system both could be utilised on Kn processors.

4. Constrained Optimisation

In this section we will consider the solution of constrained optimization problems on parallel processing systems.

The problem under consideration is therefore,

$$\begin{array}{lll} & \text{Min } f(x) & x \in R^n \\ \text{s.t.} & e_i(x) = 0 & i = 1, \ldots, E \\ & h_j(x) \geq 0 & j = 1, \ldots, J \end{array} \tag{4.1}$$

An early approach to these problems involved the introduction of penalty or barrier functions. The simplest penalty function being defined as

$$P_1(x,r) = f(x) + \frac{1}{r}\left(\sum_{i=1}^{E} e_i^2(x) + \sum_{j=1}^{J} (h_j(x))_-^2 \right) \tag{4.2}$$

where the ($)_-$indicates that the term is only included when $h_j(x) < 0$.

A barrier function method is restricted to inequality constraints and requires a feasible starting point. The barrier function is usually defined as

$$B(x,r) = f(x) - r \sum_j \log(+h_j(x)) \tag{4.3}$$

In the original algorithms Fiacco and McCormick's S.U.M.T. method was used in which $P_1(x,r)$ or $B(x,r)$ was minimised for a sequence of reducing values of r. The starting point for the next value of r was determined using the solution at previous values, so that the change in x needed, decreased rapidly with r.

It was later realised that minimising the Penalty (or Barrier) function exactly at each value of r was unnecessary and then recursive (sequential) quadratic programming codes became popular. These utilise a quadratic approximation to the Hessian of the Lagrangian and linear approximations to the constraints and decrease r at each (most) iterations. In most of these codes the quadratic approximation to the Hessian is updated using the variable metric principle so they require

accurate estimates of gradient of the function and the Jacobian of the constraints. Three popular algorithms of this type are

OPXRQP (Bartholomew-Biggs(1975))
VMCON (Powell (1977))
VMCWD (Chamberlain et al (1982))

These algorithms continue to perform as expected if analytic first derivatives are replaced by doublet automatic differentiation (Price (1987)). At about the same time we were experiencing difficulty solving space satellite trajectory optimization problems with these codes as the constraints were so nonlinear, so Brown (1986) devised a series of test problems with highly nonlinear constraints. On these problems the recursive quadratic programming codes consistently failed but an ODE method based on the implicit Euler algorithm (CONIMP) was consistently successful. This used the exact second derivatives of both objective function and constraints.

Given the success of the truncated Newton method at solving unconstrained problems. Price (1987) experimented with solving these problems by applying that algorithm to the Di Pillo and Grippo (1979) exact penalty function

$$P(x,\lambda) = f - \lambda^T e + \frac{1}{r} e^T e + \beta v^T v$$

$$\text{where } v = N^T N\lambda - N^T \nabla f \qquad (4.5)$$

$$\text{and } N_{ij} = \partial e_i / \partial x_j .$$

This penalty function has the property that its mininum occurs at the solution of the equality constrained problem for all r small enough and β big enough. He showed that if the Hessian of the term $v^T v$ is approximated in the least squares sense by $2V^T V$

$$V_{ij} = \partial v_i / \partial x_j$$

then the Hessian of $P(x,\lambda)$ can be adequately calculated from the second derivatives of the functions and constraints. He used a crude early form of automatic differentiation to obtain his results but the

algorithm LARDPG solved all of the eight highly nonlinearly constrained problems tested.

Code	Success	Fail
LARDPG	8	0
CONIMP	8	0
OPXRQP	3	5
VMCON	3	5
VMCWD	3	5

Table 20. A Comparison of Success Rates

A typical problem of this type on which the RQP methods fail is :

Objective function

$$F(\underline{x}) = \sum_{i=1}^{7} 100(x_{i+1}-x_i^{\,2})^2 + (1-x_i)^2 + 90(x_{i+3}-x_{i+2}^2)^2 +$$

$$(1-x_{i+2})^2 + 10.1[x_{i+1}-1)^2 + (x_{i+3}-1)^2] + 19.8(x_{i+1}-1)(x_{i+3}-1)$$

Constraints

$$x_1x_2 = 1.0$$
$$x_1x_2x_3x_4 = 1.0$$
$$x_1x_2x_3x_4x_5x_6 = 1.0$$
$$x_1x_2x_3x_4x_5x_6x_7x_8 = 1.0$$
$$x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10} = 1.0$$

Initial Point

$$\underline{x}_0 = (-2,\ -½,\ 3,\ 1/3,\ -4,\ -¼,\ 5,\ 1/5,\ -6,\ -1/6)^T$$

Solution

$$\underline{x}^* = (1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1)^T$$
$$F(\underline{x}^*) = 0.0$$

The detailed results of these five algorithms on this problem are given below. Here NF is the number of function/constraint evaluations and NGF the number of (second) derivative evaluations. FV the function value at the final point and VC the constraint violation.

Code	NF	NGF	FV	VC	CPU
LARDPG	60	38	6.8×10^{-9}	9×10^{-12}	110.53
CONIMP	56	332	4.5×10^{-27}	2×10^{-17}	2.03
OPXRQP	34	04	3.13×10^{5}	5×10^{-12}	0.54
VMCON	68	02	3.13×10^{5}	3.10^{-10}	1.60
VMCWD	06	06	1.24×10^{5}	9×10^{1}	1.17

Table 21. The Relative Performance of Various Codes

The automatic differentiation of the function and constraints could now be performed with sparse triplet arithmetic or even by reverse differentiation and the Hessian calculations could then benefit by both being performed in parallel and using the partially separable nature of $P(x,\lambda)$ (4.5), so the results of the LARPDG algorithm would be very much faster.

As a consequence of these results Bartholomew-Biggs devised an algorithm OPALQP (1987) as an improvement on OPXRQP, this is based on the function

$$P(x,r) = f(x) + \frac{1}{r}\left(\sum_{i=1}^{E}\left(e_i(x) - \frac{r}{2}\lambda_i\right)^2 + \sum_{j=1}^{J}\mathrm{Min}\left(0,h_j(x) - \frac{r}{2}\lambda_j\right)^2\right) \tag{4.6}$$

where λ denotes the current estimate of the Lagrange Multipliers. This algorithm works very well for small problems and is based on the

iteration

$$\Delta x = B^{-1}(A^T p - \nabla f)$$
$$\text{where } \left(\frac{r}{2} I + AB^{-1}A^T \right) p = AB^{-1}\nabla f - c + \frac{r}{2}\lambda \,. \tag{4.7}$$

An analysis of the behaviour of this algorithm showed that as the problem size increased 90% of the computational effort within the algorithm was spent in each iteration in forming the product $AB^{-1}A^T$ which requires $2En^2$ operations. This is a large overhead as the dimension increases and one that can not really be speeded up by doing parallel operations on our transputer network, due to the large amount of communication between processors that would be required. It has been our experience that it takes 3.7 times as long to send one floating point number between processors than to perform one numerical operation, it was therefore decided not to attempt to run OPALQP in parallel on that system. It would of course be a very efficient operation to do in parallel on a machine on which the communications were faster, relative to the arithmetic, as demonstrated by Conforti (1989).

Having rejected OPALQP for implementation on the transputer network, we looked for a variant designed to run on larger problems and selected MINISH Gould (1986) this uses the simplest penalty function (4.2) and applies the S.U.M.T. approach.

In particular given r_0 and r_{min} s.t. $0 < r_{min} < r_0$

set $r = r_0$ then UNTIL $r < r_{min}$
Solve Min $P(x,r)$
Set $r = r \times 10^{-2}$

with default values of r_0 and r_{min} being 10^{-1} and 10^{-12} respectively and the minimisation of $P(x,r)$ being terminated when $||\nabla P||_2 \leq 10^{-15}/r$. To obtain the results given later this terminated criteria was changed to $||\nabla P||_2 \leq 10^{-10}/r$.

The minimisation is performed by Newton's method. The equations we solved were

$$\begin{pmatrix} B & A^T \\ A & -rI \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta\lambda \end{pmatrix} = \begin{pmatrix} -\nabla f + A^T\lambda \\ e - r\lambda \end{pmatrix} \tag{4.8}$$

which is of course larger in dimension than that used in OPALQP, but does not involve the large matrix multiplication. In our implementation we used the BFGS variable metric method to update B dividing this by rows over the transputer network and replaced the direct equation solver suggested by Gould by the iterative CGS method of Sonneveld (1986) which does not depend on the matrix being symmetric or positive definite and only uses the matrix within matrix/vector products that can be performed in parallel.

Our network contained up to 14 transputers and was configured as a tree containing one transputer directly connected to the root, three transputers in the next level and nine in the final level. As each transputer only has four links it can only communicate with four other transputers and in our tree this is one transputer at the higher level and three at the lower. It is therefore logical to perform experiments with P = 4 and 14.

4.2 Results

Both OPALQP and MINISH were implemented in 3L parallel FORTRAN and run on a network of 14 T800 transputers. Both algorithms were run on two test problems:

<u>Problem 1</u>

$$\text{Min} \quad \tfrac{1}{2} \sum_{i=1}^{k-2} (x_{k+i+1} - x_{k+i})^2$$

$$\begin{aligned} \text{Subject to } & x_{k+i} - x_{i+1} + x_i = 0 && i = 1, \ldots, k-1 \\ & \alpha_i \le x_i \le \alpha_{i+1} && i = 1, \ldots, k \\ & 0.4(\alpha_{i+2} - \alpha_i) \le x_{k+i} \le 0.6\,(\alpha_{i+2} - \alpha_i) && i = 1, \ldots, k-1 \end{aligned}$$

$$\text{where } \alpha_i = 1.0 + 1.01^{i-1}, \qquad 1 \le i \le k+1$$

k	No. of Var's	No. of Const.
10	19	47
20	39	97
30	59	147

Problem 2

$$\text{Min } f(x,y,u) = \tfrac{1}{2} \sum_{t=0}^{k} x_t^{\,2}$$

Subject to $x_{t+1} = x_t + 0.2y_t$

$$y_{t+1} = y_t - 0.01y_t^{\,2} - 0.004x_t + 0.2u_t$$

$$-0.2 \le u_t \le 0.2$$

$$y_t \ge -1.0$$

$x_0 = 10,\ y_0 = 0,\ y_k = 0$ for $t = 0, \ldots, k-1$

k	No. of Var's	No. of const.
10	32	53
20	62	103
30	92	153

The following results are for each algorithm run in a sequential fashion on a single transputer.

	Problem 1	NF	NG	F	Time (Secs)
OPALQP	k=10	15	13	1.639×10^{-13}	3.07
	k=20	30	28	3.080×10^{-14}	34.69
	k=30	66	45	1.729×10^{-19}	152.839
MINISH	k-10	37	27	2.846×10^{-18}	8.64
	k=20	58	43	7.833×10^{-19}	110.08
	k=30	149	71	2.616×10^{-10}	813.15
	Problem 2				
OPALQP	k=10	28	28	550.0	42.83
	k=20	53	46	1050.0	484.82
	k=30	61	55	1550.0	2121.36
MINISH	k=10	54	29	550.0	108.79
	k=20	81	34	1050.0	847.32
	k=30	95	51	1549.9	4872.36

Table 22. Comparison of OPALQP and MINISH

From these results it can be seen that OPALQP solves the test problems in around a third of the time that MINISH takes. No doubt due to the fact the MINISH adopts the S.U.M.T. strategy. However as explained earlier MINISH has more scope for speed up when adapted for use on a transputer network. It was not felt that significant improvement could be gained by parallelising OPALQP but it was hoped that parallelising MINISH would make it more competitive, particularly for larger problems.

MINISH was therefore adapted to perform the BFGS update and to solve the Newton equations in parallel on a network of four and then 14 transputers. These adaptations gave the following results.

Problem 1	NF	NG	F	Time (Secs)
k=10	33	27	1.050×10^{-23}	6.94
k=20	58	43	7.844×10^{-19}	61.29
k=30	149	71	2.616×10^{-10}	379.30
Problem 2				
k=10	54	29	550.0	62.31
k=20	84	35	1050.0	475.05
k=30	95	51	1549.9	2317.56

Table 23. (4 Transputers)

Problem 1	NF	NG	F	Time (Secs)
k=10	33	27	2.819×10^{-18}	9.24
k=20	59	43	8.248×10^{-19}	74.43
k=30	149	71	2.616×10^{-10}	342.62
Problem 2				
k=10	54	29	550.0	100.21
k=20	81	34	1050.0	512.14
k=30	95	51	1549.9	2179.43

Table 24. (14 Transputers)

By comparing the results in Tables 1 and 2, it can be seen that there was a speed up factor of around 2 when going from a single transputer to 4 transputers on both problems.

However unfortunately when the times for MINISH on four transputers are compared to the times for OPALQP on a single transputer given in Table 1, OPALQP was still quicker on each version of the problems.

On a 14 transputer network for values of $k=10$ and 20, both problems were too small for any advantage to be gained from parallelisation. In fact the amount of communication required to send one fourteenth of the data around the network, out-weighed any savings in the computation. This resulted in slower times on 14 transputers than on 4 transputers, on both problems, except when $k=30$, when a minor speed up was achieved on both problems.

4.3 Conclusions

The modified version of MINISH described above would work very effectively on a parallel processing machine on which communication time for moving floating point numbers between processors is considerably faster than the time needed to perform an arithmetic step.

The same might be true of a modified version of OPALQP that also utilised CGS and performed the calculation $AB^{-1}A^{T}z$ in three stages $u = A^{T}z$, $y = B_u^{-1}$, $v = Ay$ with B^{-1} updated not B. Each of these multiplication could be performed efficiently in parallel on a machine with fast communications.

If the problem has very nonlinear constraints then the minimisation of 4.5 is preferable as it overcomes the difficulties due to the nonlinearities and can be done effectively by the truncated Newton method.

In each case automatic differentiation can be performed to obtain the necessary derivatives. This can utilise both the parallelism due to partial separability and if the second derivatives are used the parallelism in the reverse differentiation stage.

5. Pure Speculation?

In section 2 we have shown that any function calculation may be represented by a task graph and illustrated this concept with the graph for Rosenbrock's function. We then showed that the operations needed for reverse automatic differentiation were representable by a mirror image of the same graph. Again we noted that these graphs were linked by the nonlinearities in the objective function and that a reduced graph could be used to link the coordinate values with the gradient values and that by extending this to $\nabla f^T d$ and reversing this graph we obtain a graph for $\nabla^2 fd$.

The proposal that this graph might be used to calculate the Newton step without forming the Hessian appears in Griewank (1989b).

Let us consider what might be involved.

We have seen that for Rosenbrock's function the operations and the task graph are as given in Figure 6.

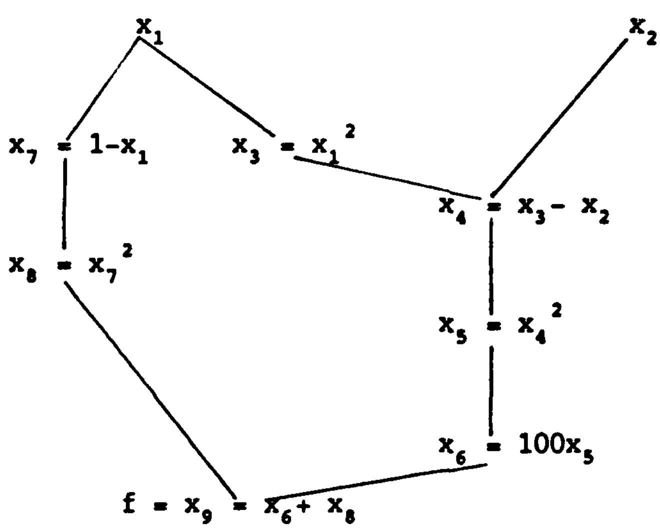


Figure 6. Task graph of Rosenbrocks Function

and that using reverse differentiation we can calculate the gradients as shown in Figure 7.

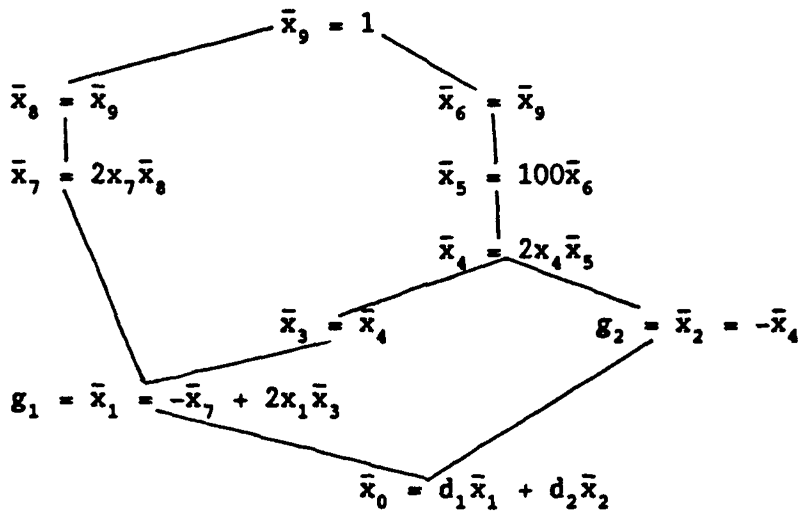


Figure 7. Reverse task graph for the gradient vector

It will be noted that the elementary operations of add, subtract and multiply appearing in f contribute a fork in ∇f while those x_i appearing nonlinearly in f also appear in ∇f. We note that until these x_i appear in the gradient tree the $\bar{x}_j$ are constant and independent of x.

Now linking these two graphs together through the nonlinearities we obtain Figure 8.

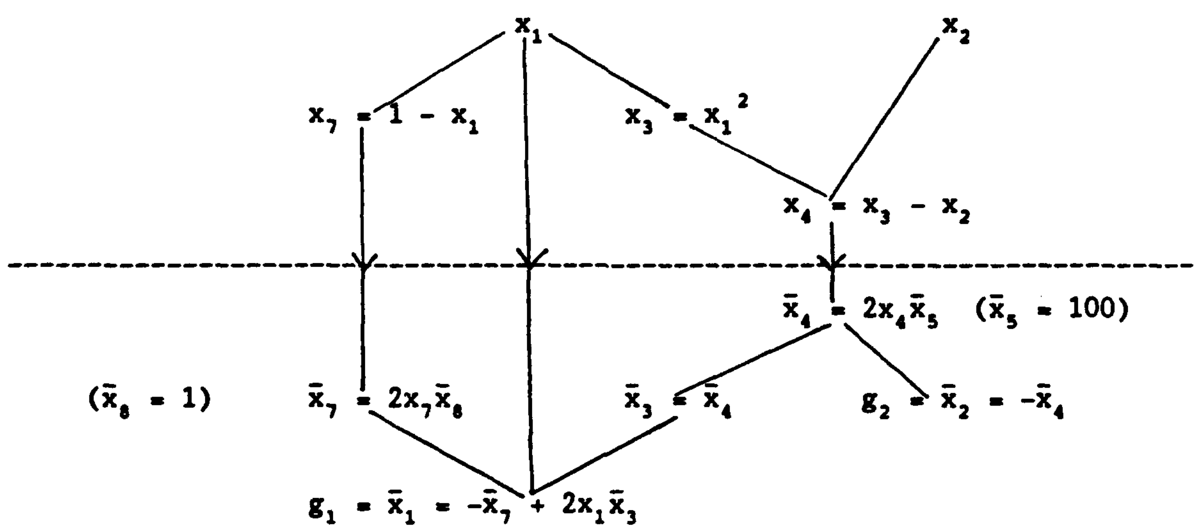
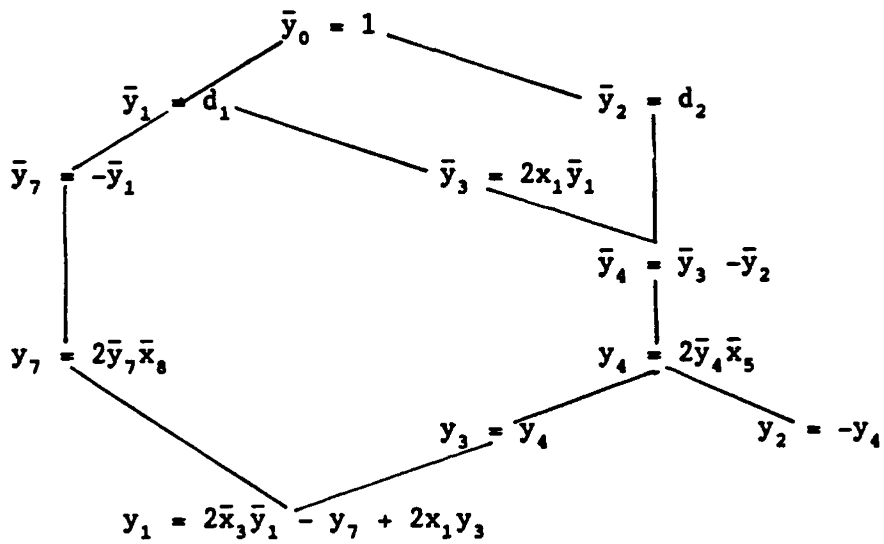


Figure 8. A direct task graph for the gradient vector

This emphasises the symmetric nature of the graph.

If we now perform reverse differentiation on this graph extended to $\bar{x}_0 = d_1\bar{x}_1 + d_2\bar{x}_2$ and representing the variables in the graph of ∇fd by y_i and $\bar{y}_i$ when it corresponds to x_i and $\bar{x}_i$ respectively we obtain Figure 9.



<u>Figure 9. Task graph for the directional second derivative</u>

where $\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \nabla^2 f \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$.

We note that the structure for this graph is identical to that for ∇f but that the operations at the nodes are slightly different. In looking for the Newton step of this problem we may treat any x_i, $\bar{x}_i$ remaining in the calculation as constants and can now represent the graph as a sparse matrix. By ordering the variables as

$\bar{y}_1\ \bar{y}_2\ \bar{y}_3\ \bar{y}_4\ \bar{y}_7\ y_7\ y_4\ y_3\ y_2\ y_1$

we obtain the sparse system.

$$
\begin{bmatrix}
1 & & & & & & & & & \\
 & 1 & & & & & & & & \\
-2x_1 & & 1 & & & & & & & \\
 & +1 & -1 & 1 & & & & & & \\
+1 & & & & 1 & & & & & \\
 & & & & -2\bar{x}_8 & 1 & & & & \\
 & & & -2\bar{x}_5 & & & 1 & & & \\
 & & & & & & -1 & 1 & & \\
 & & & & & & +1 & & 1 & \\
-2\bar{x}_3 & & & & & +1 & & -2x_1 & & 1
\end{bmatrix}
\begin{bmatrix}
\bar{y}_1 \\ \bar{y}_2 \\ \bar{y}_3 \\ \bar{y}_4 \\ \bar{y}_7 \\ y_7 \\ y_4 \\ y_3 \\ y_2 \\ y_1
\end{bmatrix}
=
\begin{bmatrix}
d_1 \\ d_2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0
\end{bmatrix}
$$

Figure 10. The Hessian Task matrix of Rosenbrock's function

This representation stresses the lower triangular form of the sparse matrix and its symmetry about the antidiagonal which appears to be a property of all "Hessian Task" matrices.

The forward calculation is:-

given $\underline{d}$ calculate the value of $\nabla^2 fd = (y_1 y_2)$.

This is obviously straight forward from the above matrix.

The Newton calculation in contrast is:-

given $y_1 = -g_1$ and $y_2 = -g_2$;

calculate the values of d_1 and d_2.

To formulate this calculation in matrix terms it is easiest to augment the above matrix with a further n rows and columns (so remembering $n = 2$ for Rosenbrock's function) we obtain the matrix shown in Figure 11.

$$
\begin{array}{c}
\\ \\ \\ \\ \\ \\ \\ \\ \\ \\
\text{EI}\Big\{ \\
\\
\end{array}
\left[\begin{array}{cccccccccc|cc}
1 & & & & & & & & & & -1 & \\
 & 1 & & & & & & & & & & -1 \\
-2x_1 & & 1 & & & & & & & & & \\
 & +1 & -1 & 1 & & & & & & & & \\
1 & & & & 1 & & & & & & & \\
 & & & & -2\bar{x}_8 & 1 & & & & & & \\
 & & & -2\bar{x}_5 & & & 1 & & & & & \\
 & & & & & & -1 & 1 & & & & \\
 & & & & & & 1 & & 1 & & & \\
-2\bar{x}_3 & & & & & 1 & & -2x_1 & & 1 & & \\
\hline
 & & & & & & & & & -1 & & \\
 & & & & & & & & -1 & & & \\
\end{array}\right]
\left[\begin{array}{c}
\bar{y}_1 \\ \bar{y}_2 \\ \bar{y}_3 \\ \bar{y}_4 \\ \bar{y}_7 \\ \bar{\bar{y}}_7 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ d_1 \\ d_2
\end{array}\right]
\left[\begin{array}{c}
0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ g_1 \\ g_2
\end{array}\right]
$$

Figure 11. Augmented Hessian Task matrix for Rosenbrock's function.

This matrix is an echelon form, with echelon index (EI) of $n = 2$, which we may write as

$$
\begin{bmatrix} L & B \\ C & 0 \end{bmatrix} \begin{bmatrix} v \\ d \end{bmatrix} = \begin{bmatrix} 0 \\ g \end{bmatrix}
$$

$$
\begin{array}{lll}
\text{i.e.} & Lv + Bd = 0 & \rightarrow v = -L^{-1}Bd \\
 & Cv = g & \\
\rightarrow & CL^{-1}Bd = -g &
\end{array}
$$

So by implication $\nabla^2 f = CL^{-1}B$ i.e. the Hessian Matrix is the Shur Complement of the augmented matrix.

In Dixon & Maany (1987) the operations needed to solve an echelon matrix of this type were analysed and shown to be dominated by

$$EI(NNZ) + 1/6(EI)^3$$

the first term being the calculation of the Shur complement by EI parallel steps and the second the solution of the Shur Complement equation which by definition is symmetric. If EI = n this operations count is precisely what would be expected to first form and then solve the Hessian matrix.

However if we could sort the augmented matrix into an echelon form with EI < n the operational cost would be greatly reduced. Dixon & Maany (1987) also describe a heuristic sort aimed at finding an equivalent matrix with a low EI.

In that paper they applied their echelon method to the non-symmetric Grenoble matrices in the Harwell set obtainable from Iain Duff and found that the echelon indices obtained were remarkably small.

Grenoble	n	115	185	216	343	512	1107
Matrices	EI	31	58	18	49	32	83

The formation of the Shur complement for these sorted matrices followed by solving the equations using its inverse was very efficient. A comparison was performed against a conjugate gradient algorithm with and without diagonal preconditioning, and also with MA28 and F04QAF.

Nonsymmetric Grenoble Problem	CG	PCG	MA28	F04QAF	Echelon Method
115	3	3	0.4	3	0.8
185	29	28	2.3	32	2.6
216 A	11	12	1.8	9	0.8
216 B	87 F	86 F	1.4 F	0.9 F	0.9 F
343	27	28	4.1	15.6	3.28
512	54	56	10.3	32.4	3.38
1107	521 F	519 F	133 F	684 F	20.97

Table 25. The performance of the echelon method on the Grenoble test set.

We were quite excited with these results but unfortunately the method need not always be stable (Duff & Reid, private communication). Indeed our results indicated it was always unstable if the original matrix is symmetric and positive definite. It did indeed fail on all the symmetric versions of the Grenoble problem; we did however complete our tests on these problems and for interest the results are given below.

Symmetric Grenoble Problem	CG	PCG	ILU/ CG	MA28	F04QAF
115	1.61	1.64	1.51	0.88	1.11
216	11.27	6.77	4.04	5.2	5.47
343	13.57	9.39	7.38	26.24	7.19
512	24.90	15.80	12.01	64.3	12.77
1107	59.15	37.64	31.55	S	25.04

Table 26. A comparison of the relative performance of a number of codes on the symmetric Grenoble problem

Here S indicates that too much store was required for the system being used.

Returning now to the task matrix of Rosenbrock's function the natural form of which already has an EI = 2, we found to our surprise that our echelon sort found an ordering with EI = 1.

$$
\left[\begin{array}{ccccccccccc|c}
1 & & & & & & & & & & & \\
 & 1 & & & & & & & & & & \\
 & & 1 & & & & & & & & & \frac{-1}{2x_1} \\
 & & -1 & 1 & & & & & & & & \\
 & & 1 & & 1 & & & & & & & \\
 & & & & -2\bar{x}_8 & 1 & & & & & & \\
 & 1 & & & & & 1 & & & & & \\
 & & & & & & -\frac{1}{2\bar{x}_5} & 1 & & & & \\
 & & & & & & & 1 & 1 & & & -1 \\
 & & & & & & & & -1 & 1 & & \\
 & & & & & & 1 & & & & 1 & \\
\hline
1 & & -2\bar{x}_3 & & & 1 & & & & & -2x_1 &
\end{array}\right]
\left[\begin{array}{c}
y_1 \\ y_2 \\ \bar{y}_1 \\ d_1 \\ \bar{y}_7 \\ y_7 \\ y_4 \\ \bar{y}_4 \\ \bar{y}_2 \\ d_2 \\ y_3 \\ \bar{y}_3
\end{array}\right]
=
\left[\begin{array}{c}
-g_1 \\ -g_2 \\ \hdashline \\ \\ \\ \\ 0 \\ \\ \\ \\ \\ \\
\end{array}\right]
$$

Figure 12. The resorted form of the Newton Task graph equations of Rosenbrock's function

We note that if we solve this by the Shur Complement method we reduce both parts of the cost. We note too that it is unstable near $x_1 = 0$.

Effectively this matrix implies that if we take $\bar{y}_3$ as the independent variable then we can compute all the variables and are left with one equation, that at node y_1, to determine $\bar{y}_3$. Note we could interchange the last two rows of the permutated matrix and end with node y_3 determining the value of $\bar{y}_3$ which seems to preserve the symmetry better.

Preliminary investigation indicates that an echelon sort can reduce the echelon index of the augmented task graph below n and thus reduce the cost of solving the task graph below that of forming and solving the Hessian matrix.

6. Conclusions

In this report we have demonstrated

(1) that the use of defined data types and operator overlays in Ada allows optimisation codes to be written so that they reflect the basic algorithm.

(2) that introducing sparse doublet and sparse triplet data types enables both the gradient and Hessian of partially separable objective functions to be calculated very efficiently. This result extends naturally to the Jacobian of constraints and if necessary to their Hessians.

(3) that the use of automatic differentiation combines naturally with the truncated Newton method to efficiently solve large unconstrained optimization algorithms.

(4) that the use of "accurate" dot products and matrix vector products greatly speeds up large scale optimization.

(5) that automatic differentiation of partially separable functions can be performed effectively using concurrent tasking in Ada on the Sequent Balance.

(6) that to perform the linear algebra within the truncated Newton algorithm effectively, in parallel, requires a machine with faster communications relative to computation than that available on the Sequence Balance or the transputer network.

(7) that the linear algebra and truncated Newton method performed effectively on the ICL/DAP machine with 4096 processors.

(8) that automatic differentiation combines effectively with RQP algorithms for the sequential solution of contrained problems.

(9) that when the constraints are too nonlinear for RQP algorithms to be effective automatic differentiation combined with a truncated Newton algorithm can obtain the solution when applied to the Di Pillo Grippo exact penalty function.

(10) that the linear algebra within Gould's MINISH algorithm can be simply divided into parallel tasks and should be effective on a machine with a faster communication time than our transputer network.

(11) that the use of reverse differentiation should be even faster than sparse forward automatic differentiation for calculating first derivatives, and can be performed in parallel to calculate second derivatives, and may lead to techniques that can calculate the Newton step cheaper than the Hessian matrix.

7. Acknowledgements

This report includes results funded from a number of sources. In particular we would like to acknowledge support from

(1) US Army Contract No.DAJA45-87-C-0038 and the British National Advisory Board that together funded the research of Maany and Mohseninia 1987-1989 and Price 1990.

(2) SERC which funded the research of Patel, Brown, Ducksbury and Mills.

(3) British Gas which funded the research of Parkhurst.

(4) The Italian CNR which funded the research of Vespucci

all of which featured within the report.

Additional financial support for related research was received from CEC IT task force, British Aerospace, British Council and Rolls Royce.

8. References

1. Bartholomew-Biggs, M. C., (1975), "An improved implementation of Recursive Quadratic Programming Methods for Constrained Optimization", Hatfield Polytechnic, NOC, TR105.
2. Bartholomew-Biggs, M. C., (1987), "Recursive Quadratic Programming Methods based on the Augmented Lagrangian", Mathematical Programming Study, 31, pp 21-41.
3. Bartholomew-Biggs, M. C., (1990), "An Introduction to Numerical Computation using Ada", Hatfield Polytechnic, NOC TR235.

4. Bartholomev-Biggs, M. C., (1989), "Automatic Differentiation and Constrained Optimization", from "Three papers on Automatic Differentiation presented at the IFAC Symposium on Dynamic Modelling & Control of National Economies, July 1989, Edinburgh, Scotland", Hatfield Polytechnic, NOC TR223.
5. Brown, A. A., (1987), "Optimisation Methods involving the Solution of Ordinary Differential Equations", Hatfield Polytechnic, PhD Thesis.
6. Byrd, R. H., Schnabel, R. B., Shultz, G. A. (1988), "Parallel Quasi Newton Methods for unconstrained optimization", University of Colorado, CU-CS-396-88.
7. Byrne, G. D. & Hindmarsh, A. C. (1987) Review Article: "Stiff ODE Solvers: A Review of Current and Coming Attractions", Journal of Computational Physics 70, 1987, pp 1-62.
8. Chamberlain, R. M., Powell, M. J. D, Lemarachel, C. & Pedersen, H. C., (1982), "The Watchdog Technique for Forcing Convergence in Algorithms for Constraint Optimization", Mathematical Programming Studies 16.
9. Chazan, D., & Miranker, W. L. (1970), "A nongradient and parallel algorithm for unconstrained minimization", SIAM Journal on Control 8, 207-217.
10. Christianson, B.,(1990), "Automatic Hessians by Reverse Accumulation", NOC, Hatfield Technical Report No.TR228, April 1990
11. Conforti, D., (1989), "Performance of Nonlinear Programming Algorithms in Parallel Computing", Eds. Evans, D. J., Joubert, FG. R., Pelcro, F. J., North Holland, 1990.
12. Dayde, M. (2989), "Parallel Algorithms for Nonlinear Programming Problems", JOTA 61(1), 23-46.
13. Dayde, M., Lescrenier, M. and Toint, P.L., "A Comparison between Straetters parallel variable metric algorithm and parallel discrete Newton methods", University of Namur, 89/16.
14. Dembo, R., Eisenstat, S. C., and Steihaug, T., (1982), "Inexact Newton Methods", SIAM Journal of Numerical Analysis, Vol 19, 1982, pp 400-408.
15. Dembo, R. and Steihaug, T., (1983), "Truncated Nevton Methods for Large Scale Optimisation", Mathematical Programming, Vol.26, 1983, pp.190-212.
16. Dennis, J. E. & Torczon, V. J. (1990), "Direct search methods on parallel machines", presented at Parallel Optimisation 2, Madison, 1990.
17. Dixon, L. C. W., (1974), "Nonlinear Optimisation: A Survey of the State of the Art", Hatfield Polytechnic, NOC TR42, 1973. Published in Software for Numerical Mathematics, Academic Press, pp 193-219, ed. D. J. Evans, 1974.
18. Dixon, L. C. W. and Maany, Z. A., (1987), "The Echelon Method for the Solution of Sparse Sets of Linear Equations". Hatfield Polytechnic, NOC TR177, February 1987.
19. Dixon, L. C. W., (1987), "A Review of Parallel Methods for Solving Sets of Linear Equations and their Application Within Optimisation Algorithms", presented at the International Symposium on "Vector and Parallel Processors for Scientific Computation II" held by IBM at Rome, September 1987.
20. Dixon, L. C. W. & Maany, Z. A., (1988), "The Performance of the Truncated Newton, Conjugate Gradient Algorithm in Fortran & Ada", The Third Interim Report, US Army Contract No. DAJA45-87-C-0038, Hatfield Polytechnic.
21. Dixon, L. C. W., Maany, Z. A. & Mohseninia, M., (1989), "Experience using the Truncated Newton Method for Large Scale Optimization", The Fourth Interim Report, US Army Contract No.DAJA45-87-C-0038, Hatfield Polytechnic.

22. Dixon, L. C. W., Maany, Z. A and Mohseninia, M. (1989,1990) "Automatic Differentiation of Large Sparse Systems", presented at IFAC Symposium on Dynamic Modelling & Control of National Economies, Edinburgh, July 1989. Paper 1, Hatfield Polytechnic, NOC TR223. Published in Journal of Economic Dynamics & Control No.14(2), 1990.
23. Dixon, L. C. W. and Price, R. C., (1986,1988), "Numerical Experience with the Truncated Newton Method". Hatfield Polytechnic, NOC TR169, May 1986. Published in JOTA, Vol.56, No.2, pp 245-255, February 1988.
24. Dixon, L. C. W. and Price, R. C., (1986,1989), "The Truncated Newton Method for Sparse Unconstrained Optimisation using Automatic Differentiation", Hatfield Polytechnic, NOC TR170, October 1986; Published in JOTA, Vol.60, No.2, pp 261-275, February 1989.
25. Dixon, L. C. W. and Mills, D., (1990), "The Effect of Rounding Error on the Variable Metric Method", Hatfield Polytechnic NOC TR229, April 1990.
26. Dixon, L. C. W. and Mohseninia, M., (1987), "The use of the Extended Operations Set of Ada with Automatic Differentiation and the Truncated Newton Method", Hatfield Polytechnic NOC TR176, April 1987.
27. Dixon, L. C. W. and Mohseninia, M., (1989), "Concurrent Optimisation on the Sequent Balance 8000". Hatfield Polytechnic NOC TR226, September 1989.
28. Ducksbury, P. G., (1984), "An Investigation of the Relative Merits of Optimisation Algorithms on the ICL-DAP", Hatfield Polytechnic PhD Thesis, October 1984.
29. Fiacco, A. V. & MacCormick, G. P., (1968), "Nonlinear Programming Sequential Unconstrained Minimization Techniques", John Wiley & Sons, New York.
30. Fletcher, R. and Reeves, C., (1963), "Function minimization by conjugate gradients", The Computer Journal 6, 163-168.
31. Gould, N. I. M., (1986), "On the Accurate Determination of Search Directions for Simple Differentiable Penalty Functions", IMA JNA, 6, pp 357-372.
32. Griewank, A., (1988), "On Automatic Differentiation", in Mathematical Programming 88, Kluwer Academic Publishers, Japan.
33. Griewank, A., (1989), "Direct Calculation of Newton Steps without accumulating Jacobians", Preprint MCS-P132-0290, Argonne National Laboratory.
34. Griewank, A. and Toint, Ph.L., (1981), "On the Unconstrained Optimization of Partially Separable Functions", in Nonlinear Optimization 1981, ed. M. J. D. Powell, Part 5, pp 301-312.
35. Jha, Manoranjan, (1990), "Preliminary Results on some Parallel Linear Algebra Applications on Transputer Networks", Hatfield Polytechnic, NOC TR231, April 1990.
36. Lootzma, F. A. (1986), "Parallel Algorithms for unconstrained and constrained nonlinear optimization", Bergamo University 1986, No.5.
37. Maany, Z. A., (1989), "The Performance of the Truncated Newton Conjugate Gradient Algorithm in Fortran and Ada", Hatfield Polytechnic NOC TR210, 1989.
38. Mohseninia, M., (1989), "Parallel Automatic Differentiation in Ada Applied to the Navier Stokes Equations", from "Three papers on Automatic Differentiation presented at the IFAC Symposium on Dynamic Modelling & Control of National Economies", July 1989, Edinburg, Scotland, Hatfield Polytechnic TR No223.
39. Nelder, J. A. and Mead, R., (1965), "A Simplex method for function minimization", The Computer Journal 7: 308-313.

40. Parkhurst, S. C., (1989), "Some Experiences using Rounded Interval Analysis when Solving Sets of Linear Equations", Hatfield Polytechnic TR No217.
41. Parkhurst, S. C. (1990), "The Evaluation of Exact Numerical Jacobians using Automatic Differentiation", Hatfield Polytechnic NOC TR224, December 1990. Presented at The Eleventh Conference on Differential Equations, Dundee University, July 1990.
42. Patel, K. D., (1982), "Implementation of a Parallel (SIMD) Modified Newton Algorithm on the ICL DAP", paper presented at the Progress in the use of Vector and Array Processors Workshop, University of Bristol, September 1982.
43. Polak, E. & Ribière, G., (1969), "Note sur la convergence de methodes de directions conjuguées", Rev. Française Informat. Recherche Opérationelle, s^{e} Année, No.16, pp.35-43; (1.2).
44. Powell, M. J. D., (1977), "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations", Lecture Notes in Mathematics 630, Ed. Watson, G. A., Springer Verlag.
45. Price, R. C., (1988), "Sparse Marix Optimization using Automatic Differentiation", Hatfield Polytechnic PhD Thesis, January 1988.
46. Rall, L. B., (1981), "Automatic Differentiation: Techniques and Applications", Springer-Verlag, Berlin, Germany, 1981.
47. Schnabel, R. B., (1988), "Sequential & Parallel Methods for unconstrained optimization", University of Colorado, CU-CS-415-88.
48. Sofer, A. & Nash, S., (1990), "General-purpose Parallel Algorithm for Unconstrained Minimization", Presented at SIAM Annual Meeting, Chicago, July 1990.
49. Sonneveld, P., (1986) CGS, "A Fast Lanczos-Type Solver for Nonsymmetric Linear Systems", SIAM Journal Sci. Stat. Comp., Vol.10, No.1, pp36-52, January 1986.
50. Spedicato, E., (1976), "On a Conjecture of Dixon and other Topics in Variable Metric Methods", University of Bergamo, Italy, 1976.
51. Straeter, T. A. (1973), "A parallel variable metric optimization algorithm", NASA, TN D-7329, Langley Research Centre, Virginia.
52. Sutti, C., (1983), "Nongradient Minimization Methods for Parallel Processing Computers", JOTA 39, 465-488.
53. Torczon, V. J. (1989), "Multi-directional search; A direct Search Algorithm for Parallel Machines", Rice University, PhD Thesis, TR 90-7.
54. Van Laarhoven, P. J. M. (1985), "Parallel Variable Metric Algorithms for Unconstrained Optimisation", Mathematical Programming 33, 68-81.
55. Vespucci, M. T., (1990), "The Use of the ABS Algorithms in Truncated-Newton Methods for Nonlinear Optimisation", University of Bergamo. Presented at NATO Summer School, Il Ciocco, Italy, September 1990.